# Etude des structures de données au cœur des algorithmes 3D des jeux de type FPS

**Encadrant :**

Michel BUFFA

**Participants :**

Jean-François FOURMOND
Stéphane MARIANI
Xavier MEDIONI
Jean-François RIGHI

# Table des matières

**Avant propos**
**Introduction et présentation du sujet**
**Répartition de tâches**
**Méthode de travail**
**Moteur 3D utilisé par les programmes de démonstration de ces algorithmes**

- Le moteur 3D
- Qu'est-ce qu'un moteur 3D ?
- Le moteur lui-même
- Fonctionnalités
- Technologies utilisées
- Architecture
- Le chargeur de fichiers
- Conception de l'environnement

**Les quadtrees**

- Qu'est-ce que c'est ?
- A quoi ça sert ?
- Concrètement
- Passons à la 3D
- Clipping à l'horizon
- Optimisations possibles
  - Level Of Details
  - Optimisation du frustum culling
  - Quadtrees dynamiques
- Conclusion

**Les Octrees**

- Introduction
- Structure et méthode de création
- Utilisation
- Optimisations possibles :
- Implémentation
- Stockage des Octree / Conception de l'arbre
- Gestion des collisions
- Simulation d'éclatage de la scène
- Conclusion

**Les arbres BSP**

- Historique
- Définition
- Limitation des arbres BSP
- Construction de l'arbre BSP à partir d'un fichier monde (Compilation)
- Choix du plan de partitionnement
    - Choisir les pivots aléatoirement
    - Choisir comme pivot la première face de la liste
    - Choisir comme pivot le plan qui coupe le moins de faces
    - Choisir le pivot en vue d'obtenir un arbre équilibré
    - Conclusion sur le choix du pivot
- Initialisation de la bounding box de chaque nœud
- Parcours de l'arbre pendant l'exécution
- Programmes implémentés
    - Compilateur d'arbres BSP
    - Moteur 3D utilisant l'arbre BSP compilé
    - Conclusion sur l'utilisation des arbres BSP

**Les Portails**

- Qu'est ce qu'un portail?
- Présentation de l'algorithme
- Notions mises en jeux dans l'algorithme
- Les pièces, les portails et la récurrence
- Comment représenter une pièce et un portail ?
- Polygones et portails
- Visibilité clipping vs Bouding Box
- Première passe: extraction géométrique de 4 points du plan qui sont les points d'accroche du frustum
- Deuxième passe: calcul des équations des 4 plans :
    - Petits rappels Mathématiques: les classes Plan, Vecteurs et Point
- Visibilité
- Le recalcul du frustum
- L'algorithme
- Ray Casting implémentation
- Optimisations et réflexions
- Conclusion et Annexes

**Synthèse sur l'étude de ces algorithmes**

**Conclusion**

# Avant-propos

Ce rapport ayant été conçu sur le wiki, il comporte des liens permettant d'accéder à nos références, nos programmes (pour Windows), et de passer rapidement d'une partie du rapport à une autre. Nous vous conseillons donc, si vous êtes sur la version papier, de préférer la version en ligne à l'adresse suivante :

http://deptinfo.unice.fr/cgi-bin/twiki/bin/edit/Minfo/AlgosJeux3DRapport.

# Introduction, présentation du sujet

Lors du cours d'introduction à la synthèse d'images de Michel Buffa nous avons surtout étudié les bases de la visualisation 3D : bases mathématiques, calculs en coordonnées homogènes, projection 3D vers 2D, algorithmes d'élimination des faces cachées, clipping 2D et 3D, gestion des éclairages, lissage des faces (Gouraud, Phong) et texture mapping.
Malheureusement, en 6 séances nous n'avons pu implémenter dans notre programme développé en TP (et mini projet) l'ensemble de ces algorithmes faute de temps.

Par ailleurs, le logiciel développé était un simple visualiseur d'objets 3D, en aucun cas un logiciel "lourd" permettant par exemple de se promener dans des univers importants tels qu'un bâtiment ou un paysage de plusieurs kilomètres.
Or la plupart des jeux vidéos ou des applications de réalité virtuelle permettent aujourd'hui d'explorer des environnement très grands. Se pose alors un problème récurent dans ces applications : étant donné que la somme des données nécessaires à la représentation tridimensionnelle de cet univers consiste en des milliers voire des millions de polygones, comment les "trier" afin de ne considérere que la partie de ces informations qui correspond à ce qui doit réellement être dessinée sur l'écran ?

La solution est algorithmique. Imaginons que l'on classe dans une simple grille 2D dont chaque case fait 10 mètres de côté les objets tridimensionnels qui composent l'univers à explorer, si on veut dessiner à l'écran ce que voit un individu, il est inutile de considérer les cases qui sont en dehors de son champ de vision. On a donc déjà un algorithme permettant de simplifier la quantité de données à traiter ! Ceci n'est qu'un exemple simple, nous verrons dans ce rapport que de nombreux cas se présentent et qu'un simple algorithme à base de grille ne suffit pas toujours !

Les quatre algorithmes les plus utilisés seront présentés, chacun portant le nom de la structure de données sur laquelle il est basée. Pour chacun des programmes de démonstration à but pédagogique ont été réalisés (Monsieur Buffa compte les utiliser l'an prochain pour illustrer son cours).

Les algorithmes étudiés sont :

- Quadtrees (Jean-François Fourmond),
- Octrees (Stéphane Mariani),
- Portails (Jean-François Righi),
- Arbres BSP (Binary Space Partition tree) (Xavier Medioni).

Par ailleurs, pour illustrer ces algorithmes, un moteur 3D a été réalisé, version beaucoup plus aboutie que celui développé en cours. Il utilise la librairie OpenGL et la librairie SDL. Il est à noter qu'en TP de synthèse d'images, nous avons développé le moteur 3D "from scratch", sans l'aide d'aucune librairie, ce qui nous a permis de bien comprendre les mécanismes internes. L'utilisation d'une librairie 3D telle qu'Open GL s'est donc trouvé facilitée. Ce moteur 3D sera détaillé dans une section dédiée.

Pour nous faire une idée des avantages et inconvénients de ces algorithmes, nous avons crée un environnement en 3D à charger par le moteur 3D. Vous verrez que certaines parties de cet environnement se prêtent mieux à certains algorithmes que d'autres. Une section de comparaison des algorithmes est présentée dans le rapport.

# Répartition de tâches

- Elaboration du Moteur 3D : Jean-Francois Fourmond
- Elaboration du chargeur de fichier du moteur 3D : Stéphane Mariani
- Elaboration d'un environnement 3D pour le chargeur : Xavier Médioni et Jean-François Righi
- Elaboration du frustum culling : Jean-Francois Fourmond et Stéphane Mariani

# Méthode de travail

Nous avons tâché de communiquer le plus possible :

- Par l'intermédiaire du Wiki avec des mises à jour quotidiennes,
  - en communiquant entre nous sur une page dédiée
  - en montrant régulièrement l'avancée de notre travail sur nos pages respectives
- Par des réunions fréquentes entre les membres de l'équipe
- Nous avons aussi rencontré Nicolas Peri, développeur de jeux vidéos sur différentes plateformes, et avons longuement communiqué avec lui, notamment sur nos algorithmes et les techniques utilisées dans les jeux actuels

# Moteur 3D utilisé par les programmes de démonstration de ces algorithmes

## Le moteur 3D

Avant tout, nous présentons ici les caractéristiques du programme que nous avons développé pour illustrer les algorithmes précités. Il s'agit d'un moteur 3D.

# Qu'est-ce qu'un moteur 3D ?

Un moteur 3D peut se définir comme un programme calculant des images affichables avec un minimum de fluidité (que l'on peut mesurer par le taux de rafraîchissement) et simulant le déplacement d'une caméra virtuelle et/ou d'objets dans un environnement en trois dimensions.

Une fois nos algorithmes de partitionnement de l'espace en tête nous avons choisi de commencer par implémenter un moteur 3D commun, qui pourrait donc être utilisé par chacun de nous. Le travail a été découpé en deux parties :

- Le moteur lui-même
- Le chargeur de fichiers

# Le moteur lui-même

Avant tout, il faut préciser que nous n'utilisons aucune optimisation d'Open GL. Par exemple, nous envoyons à la carte graphique les polygones un par un, ce qui n'est pas du tout optimal. Nicolas Peri nous a dit qu'il fallait les envoyer par paquets de plusieurs centaines pour avoir de bonnes performances, sinon cela revient à utiliser la carte en sous-régime.

Les seules optimisations qui font aller le programme plus vite sont donc nos algorithmes, que nous détailleront plus bas.

## Fonctionnalités

- Visualisation 3D à l'aide de caméras virtuelles : quand on se promène dans l'univers tridimensionnel, ce qu'on voit sur l'écran, c'est le point de vue d'une personne (comme dans les jeux de type FPS) ou d'une caméra (virtuelle) située quelque part dans l'univers, qui regarde dans une certaine direction. Cette caméra possède des caractéristiques telles que la focale, la taille de la pellicule (la taille de la fenêtre de visualisation dans notre cas), un champ de vision comme on verra plus bas.

- Ce moteur 3D permet le même type de contrôles que les jeux de type FPS :
    - on marche avec le flèches du clavier
        - en avant ou en arrière
        - en pas chassés (strafe)
    - on s'oriente avec la souris
        - horizontalement à 360° (l’utilisateur choisi son sens de rotation)

        - verticalement à 180° (on peut au maximum regarder au dessus de nous on en dessous, pas en arrière)

- 2 modes de déplacement sont proposés :
    - Un mode de déplacement dans le plan
        - C'est le même mode que dans les jeux de type FPS, on marche sur un plan horizontal qu'on choisit, et on y reste. Par exemple, si on regarde en haut et qu'on avance avec le clavier, on va marcher en avant tout en regardant en haut, mais toujours en restant au niveau du sol. Ce mode est accompagné d'une modification légère de la position de la caméra en Y selon la fonction sinus. Ainsi, la caméra monte et descend quand on avance, de manière à simuler un effet "course à pieds".
    - Un mode de déplacement dans l'espace
        - Le mode de déplacement dans le plan n'est pas très pratique pour bien observer tout le monde en 3D et les effets des algorithmes de partitionnement, c'est pourquoi nous avons rajouté un mode de déplacement dans l'espace. Dans ce mode, on est plus attaché au sol et on peut évoluer librement dans l'espace. Maintenant, quand on regarde en haut et qu'on veut avancer avec le clavier, on va monter dans l'espace . Ainsi on pourra avoir une vue de dessus globale du monde en 3D.

- Il offre une vitesse de déplacement constante : on va à la même vitesse
    - quelque soit la rapidité de l'ordinateur,
        - Pour cela l'ampleur de chaque déplacement dépend du temps écoulé entre 2 images. Par exemple un déplacement sera deux fois plus important si on est à 10 images/seconde que si on est à 20 images/seconde.
    - qu'on regarde en à la verticale en haut ou en bas, ou qu'on regarde droit devant,
    - et on ne va pas plus vite si on avance en strafant que si on avance ou on strafe seulement.

- Plusieurs modes de visualisation sont disponibles :
    - Mode "sommets" : on affiche que les sommets des polygones
    - Mode fil de fer : on affiche que le contour des polygones
    - Mode texturé : on affiche les polygones en leur plaquant des textures

- Affichage de texte à l'écran
    - Nombre d'images par seconde, de polygones affichés, informations relatives à un algorithme particulier, etc...
    - On utilise pour cela la librairie BMF_font.

## Technologies utilisées

Le moteur est écrit en C++. La librairie utilisée pour le fenêtrage est la SDL. Elle est portable et permet de gérer simplement les évènements utilisateur. Toute le rendu (texte et 3D) se fait à l'aide de l'API Open GL.

Open GL permet de gérer facilement la mise en place d'un environnement en 3D, et utilise l'accélération matérielle des cartes graphiques. Les performances sont donc bien meilleures que si on faisait tous les calculs (rotations de matrices, projection du monde 3D sur l'écran en 2D, placage de textures, etc...) nous-mêmes car ça décharge les CPU des tâches liées à l'affichage.

N'étant pas familiarisés Open GL auparavant, nous avons avant tout dû apprendre à l'utiliser.

## Architecture

Notre moteur 3D est inspiré d'un exemple de GameTutorials, mais a été totalement repensé et modifié. Il conserve toutefois des bouts de code comme l'initialisation de la SDL et d'Open GL, la gestion des évènements ou le calcul du nombre d'images par seconde.

Le moteur 3D de base est constitué d'un fichier main.cpp qui réagit aux évènements avec la SDL, d'une classe Camera, d'une classe Scene et d'une classe Coord3D.

La classe **Coord3D** correspond à une coordonnée dans l'espace (X, Y, Z)

La classe **Camera** a les attributs suivants :

- Un de type Coord3D qui représente sa position dans l'espace.
- Un autre de type Coord3D qui représente le point de l'espace que la caméra regarde (point de visée).
- Un 3e de type Coord3D, qui représente le vecteur directeur vertical.

Avec ces 3 attributs, on pourra se déplacer dans l'espace en utilisant Open GL. Les évènements clavier et souris enregistrés par la SDL modifieront la position et le point de visée de la caméra, ce qui la fera se déplacer.

La classe **Scene** s'occupe de l'affichage du monde en 3D. A chaque tour de boucle, la classe Scene dessinera le monde en 3D en fonction de la position et de l'orientation de la caméra.

# Le chargeur de fichiers

- **Introduction :**

Quand on a un grand nombre de polygones, il est difficile de les définir dans le code. De plus, on veut pouvoir changer d'environnement 3D. Ces environnements sont décrits dans des fichiers, et c'est pour cela que nous avons implémenté un chargeur permettant de les lire et de les charger en mémoire.

Nous utiliserons pour notre moteur le format de fichier obj.
En effet ayant eu l'occasion de le manipuler et donc de nous familiariser avec celui-ci

lors de l'élaboration du projet de synthèse d'image et de même correspondant bien aux exigences de notre travail nous avons jugez opportun de l'exploiter.

- **Description du format .obj :**

Voici les differents léxèmes nécessaires à la définition d'objet 3D :

- # commentaire
  La ligne est commentée jusqu'à la fin de la ligne

- g nomgroupe_numeroelementdansgroupe
  Un groupe est équivalent à un objet, il est suivi des sommets qui forment les faces ainsi que de ses faces.

- v float float float
  Un sommet (vertex) et sa position dans l'espace. Le premier sommet listé a comme index 1 et les sommets subséquents sont numérotés séquentiellement.

- f int int int ...
  Une face polygonale. Les entiers sont indéxés respectivement dans un tableau de sommets.

Dans notre fichier, les polygones contiennent trois ou quatre sommets. La spécification du fichier .obj précise que chaque face doit être plate et convexe.
Ce format de fichier 3D ne prend pas en compte la sauvegarde du matériau, du nom du matériau, de couleur et de la texture qu'il lui est associé. Ce format n'est pas de la meilleure utilité pour les jeux, mais dans notre cas, il convient très bien, nous apporterons cependant quelques modifications au format du fichier afin d'apporter à ce format les concepts de matériaux et de couleurs de l'objet.

- **Classes et Structures :**

Dans le fichier **Structs.h** :

On a la définition de la structure de **Face** :
Pour connaître le type de face il suffit de faire un test sur le booléen isAQuad qui est à vrai si la face est un quadrilatère, faux si la face est un triangle.
On a la définition de la structure de **Materiau** :
Cette structure contient les informations nécessaires à la description d'un matériau, c'est à dire un nom, un nom de fichier auquel on trouve l'image de la texture, un identifiant et une couleur de base.

La Classe **Model3D** :

Comme son nom l'indique cette classe permet de définir un modèle. Quelle est la différence entre un modèle et un objet ? Le modèle contient un tableau d'objets ainsi qu'un tableau de matériaux que l'on va utiliser avec les objets. Cette notion intervient lors du texturage. Dans le fichier .obj, chaque élément du décor est défini sous forme de groupe, d'abord la liste des sommets puis les faces qui vont former l'élément en question.
A cet élément du décor on va associer une texture (un matériau), tous ces éléments sont donc des objets définis par la classe Object3D.
On créé donc autant d'objets que de groupes. Le modèle contient au final un tableau de tous ces objets, ainsi qu'un tableau de tous les matériaux nécessaires.
On y trouve aussi une fonction Draw() permettant d'afficher le modèle.

La Classe **Object3D** :

Cette classe englobe tous les éléments nécessaires à la définition d'un objet c'est a dire d'un élément du décor ou du modèle. On y trouve donc un tableau dynamique de sommets, un tableau dynamique de faces, l'identifiant du materiau qui est l'index dans le tableau des textures.

La Classe **Loaderobj** :

Cette classe est donc le chargeur de fichier.
Pour l'explication voici le déroulement de la mise en structure.
On ouvre le fichier en lecture. On a des tableaux temporaires dans le loader, on précise bien un tableau dynamique (un vector utilisation de la STL) car on ne connaît pas d'avance le nombre de sommets que l'on va lire, et de même pour les faces.
On lit des sommets et on les stocke dans le tableau temporaire des sommets, ensuite on lit des faces et on les stocke dans le tableau temporaire des faces.
Si lorsqu'on est en train de lire des faces, une information concernant un sommet est lue (en l'occurence le caractère lu est un 'v') c'est que l'on vient de finir de lire un objet, donc un élément du modèle, auquel cas on construit un nouvel objet à partir des tableaux temporaires que l'on vient de remplir, et on ajoute cet objet ainsi créé à la liste des objets du modèle.

Petite précision technique, puisque l'on ne connaît pas d'avance le nombre d'objet dans le modèle, les conteneurs utilisés d'objets et de matériaux pour le modèle sont des vector (STL), des tableaux dynamiques. Par contre, lorsqu'on construit un objet avec des tableaux temporaires on connaît avant la taille des tableaux dynamiques, on peut par conséquent allouer la place nécessaire, c'est pourquoi la classe Object3D travaille avec des tableaux et non des tableaux dynamiques.
Les fichiers obj ne contenant pas d'information sur les matériaux on va devoir les créer et mettre en relation un objet et sa texture avec nos propres fonctions. Il y a deux fonctions qui se chargent de ça, une qui ajoute un matériaux, et une qui met en relation un objet avec un matériaux.
Nous détaillons ci-dessous la façon de procéder.

- **Fonctionnement global :**

Comment charger une scène et ou sont placés les fonctions d'appel ?
Le chargement global de l'objet (lecture des données du fichier et chargement des textures) se fait dans le fichier main.cpp, dans une fonction d'initialisation.

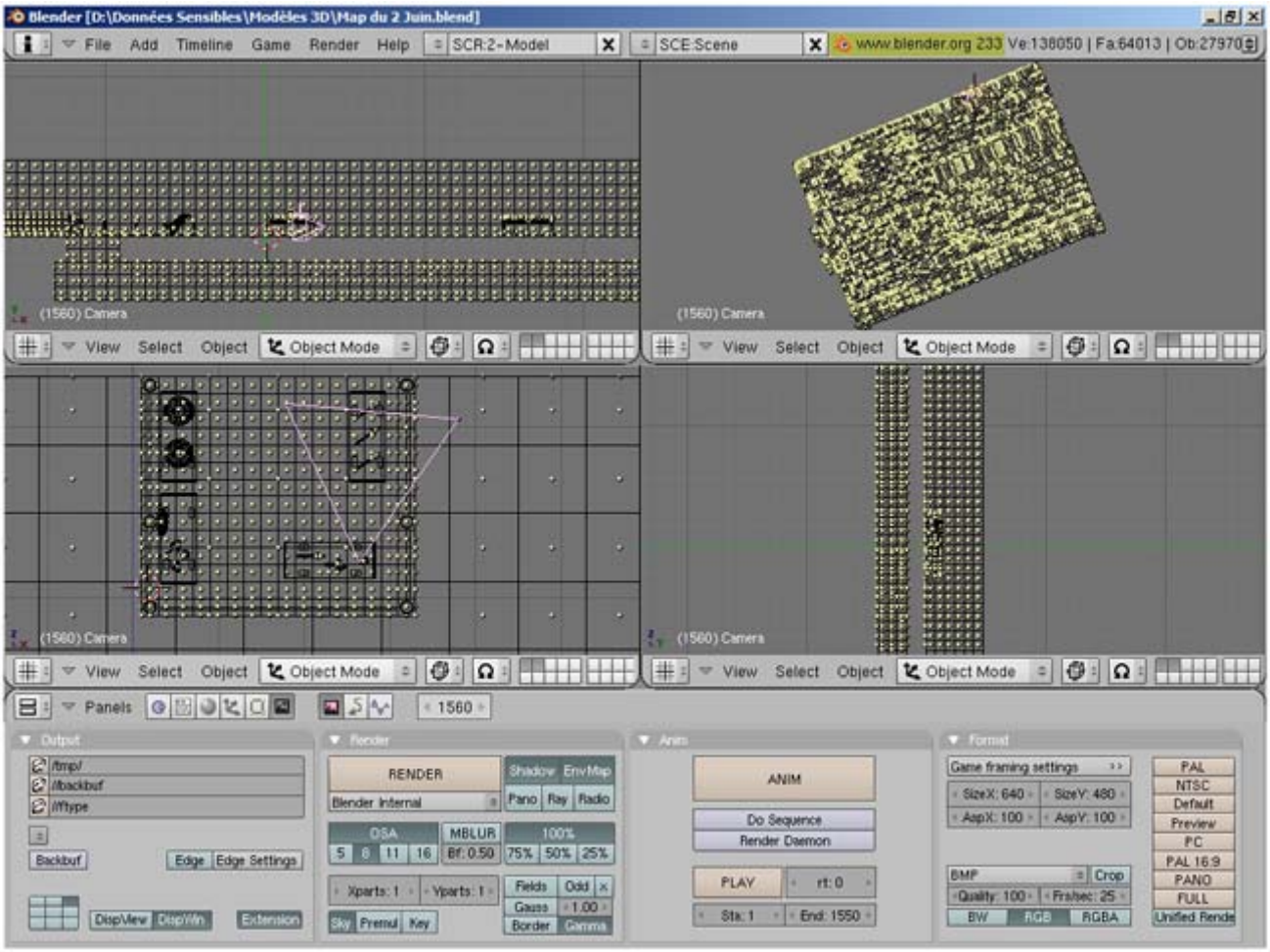
La première étape et de charger le modèle, notre monde, à partir d'un fichier dans une variable donc de type Model3D.
A présent nous possédons donc en mémoire une variable de type Model3D contenant un tableau d'Object3D ces derniers possédant donc les informations sur les sommets, les faces ainsi que le NOM de la texture dont il est composé. On rappelle qu'à un objet correspond un groupe donc une texture.
La seconde étape consiste donc à parcourir tous les objets du modèle et à associer à chaque objet un identifiant correspondant à un numéro de matériau; nous dirons qu'un matériau est caractérisé par un identifiant de matériau ainsi qu'une texture. On remarque que les textures ne sont toujours pas chargées en mémoires.
C'est le rôle de la troisième étape. L'objet Model3D possède un tableau de Matériau contenant les informations de tous les matériaux intervenant dans la scène, ou disons sinon tous les matériaux dont les objets du modèle sont constitués.

Ainsi il suffit de parcourir ce tableau de matériaux et de créé la texture à partir du nom de fichier. Il en résulte en fin de compte un tableau de texture indexé par un numéro d'identifiant et contenant les données de textures d'un matériau désigné par cet identifiant.
Chaque objet ayant un identifiant de matériau il suffit de charger lors de l'affichage la texture contenue à l'index numéro d'identifiant dans le tableau des textures.

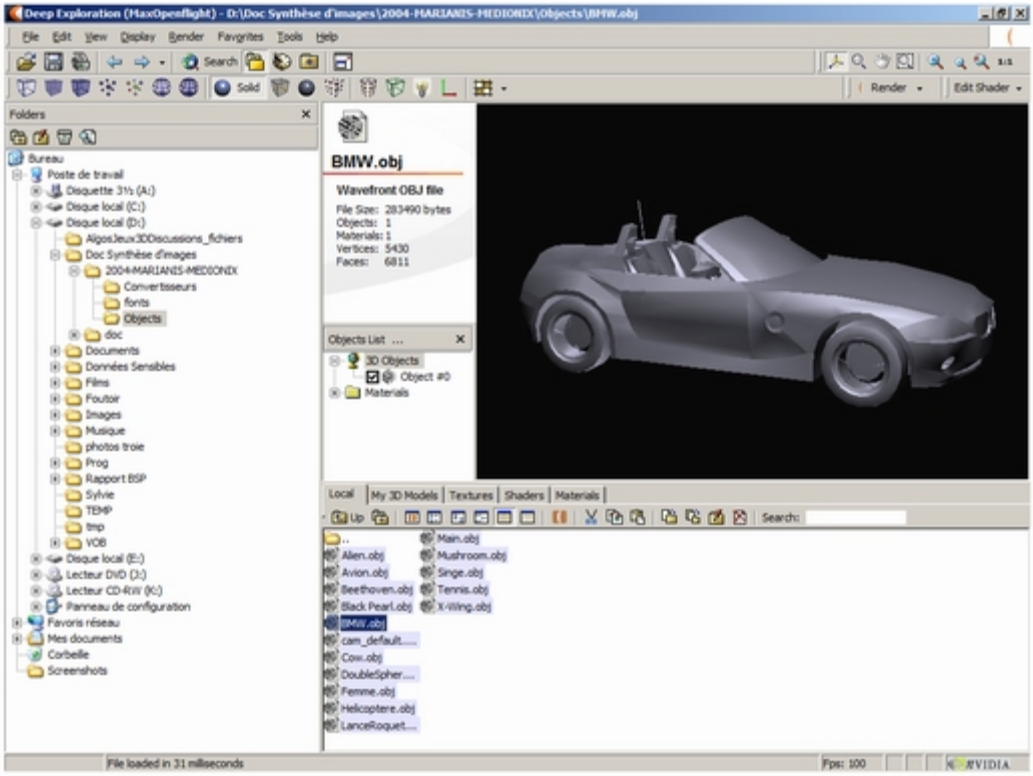
Ce moteur permet donc de charger un univers en 3D et s'y déplacer, mais tel quel dans un monde tel que celui qu'on a créé, composé de plus de 60000 polygones texturés, le nombre d'images par seconde est très bas : à peine plus de 10 images par seconde pour des machines très puissantes. Pour que ça aille plus vite, il va falloir éliminer le plus possible de polygones qui ne sont pas visibles. C'est là qu'interviendront le frustum culling et nos algorithmes.

# Conception de l’environnement

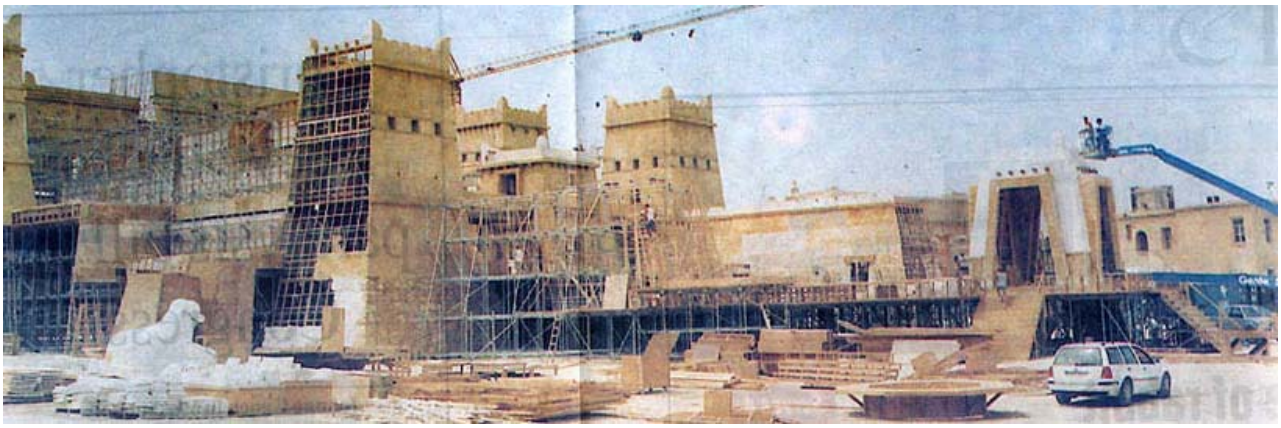
Le sujet de notre TER étant l’étude des algorithmes de partitionnement de l’espace, l’objectif était de concevoir un environnement composé d’une quantité de polygones suffisamment importante pour voir une différence conséquente lorsque la caméra virtuelle se déplace dans l’environnement non partitionné et lorsqu’elle se déplace dans l’environnement partitionné grâce à un des algorithmes étudiés.
Le logiciel utilisé pour réaliser cet environnement est Blender, un logiciel de modélisation 3D sous licence GPL.




Les opérations de conversion de formats de fichier ont été réalisées avec une version d’évaluation de Deep Exploration 3.



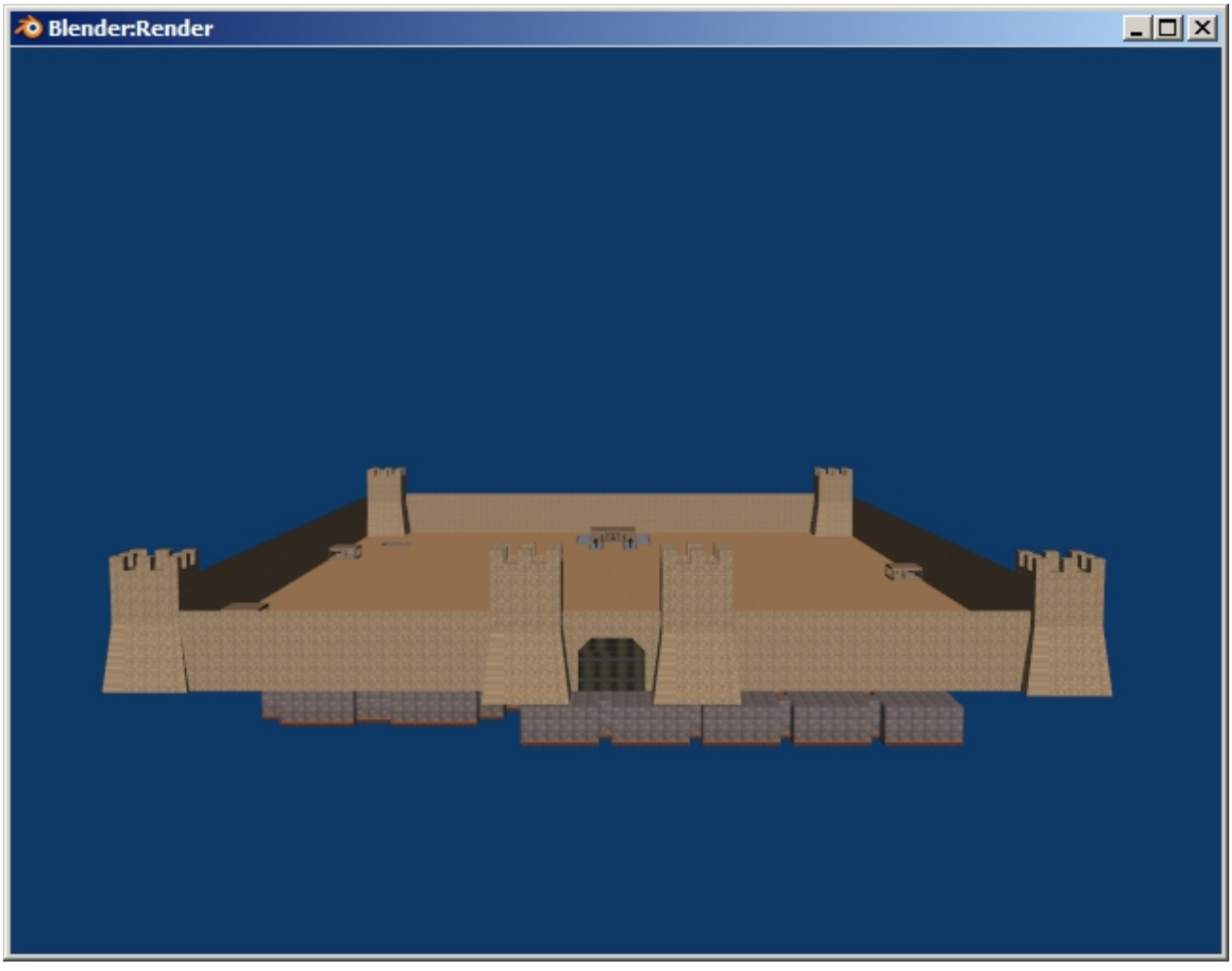

L'environnement qui a été conçu a été modelé dans le but d'être composé d'une multitude de polygones. Par exemple, au lieu de se servir d'un seul grand polygone rectangulaire pour représenter le sol, j'ai préféré utiliser une multitude de dalles mises bout à bout. Au final, l'environnement est composé de plus de 60 000 polygones. Pour pouvoir avoir un ordre de grandeur, les niveaux de Quake (Id Software) sont composés d'environ 10 000 polygones.
J'ai essayé de reconstituer un environnement similaire à celui du film « Troie » (sorti le 13 Mai en France avec Brad Pitt, Orlando Blum, Diane Kruger) dont voici certains clichés du tournage :



Le délai imposé pour rendre le TER ne m’a hélas pas permis de m’attarder trop longtemps sur la réalisation de l’environnement. Il se décompose en :

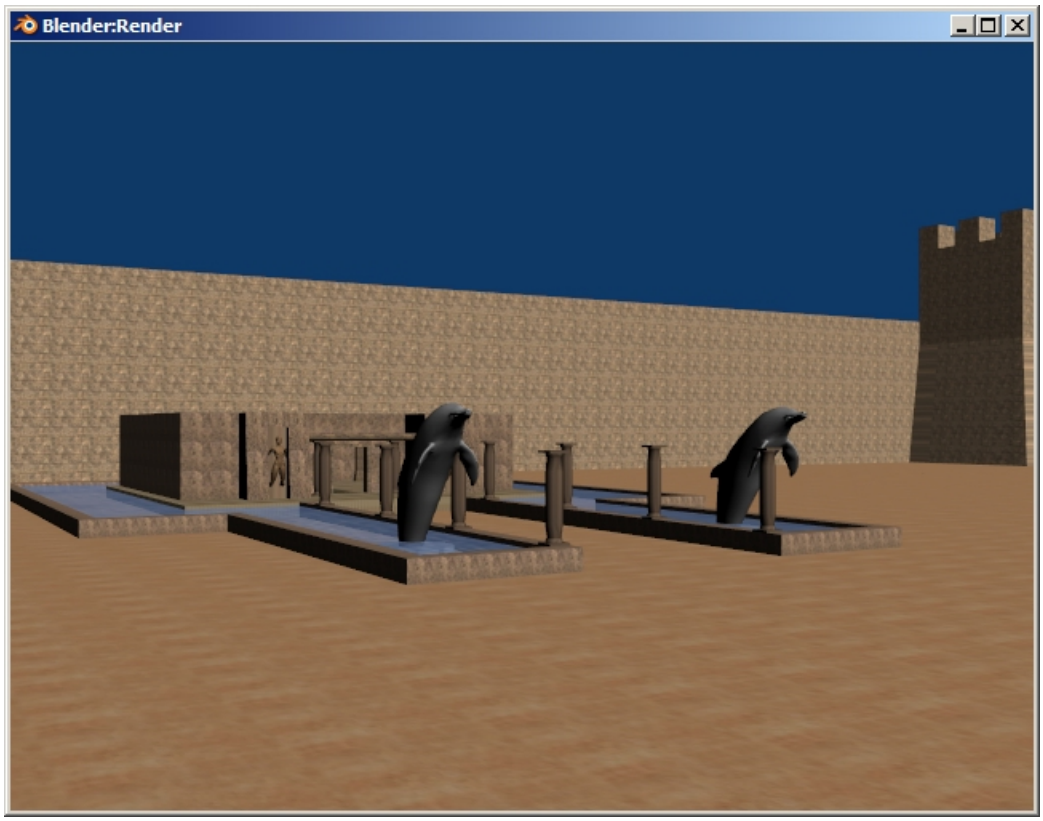
1. une muraille avec des tours afin de délimiter l’environnement
2. un temple entouré par un plan d’eau
3. des marchands d’armes
4. un labyrinthe sous-terrain

Vue d'ensemble

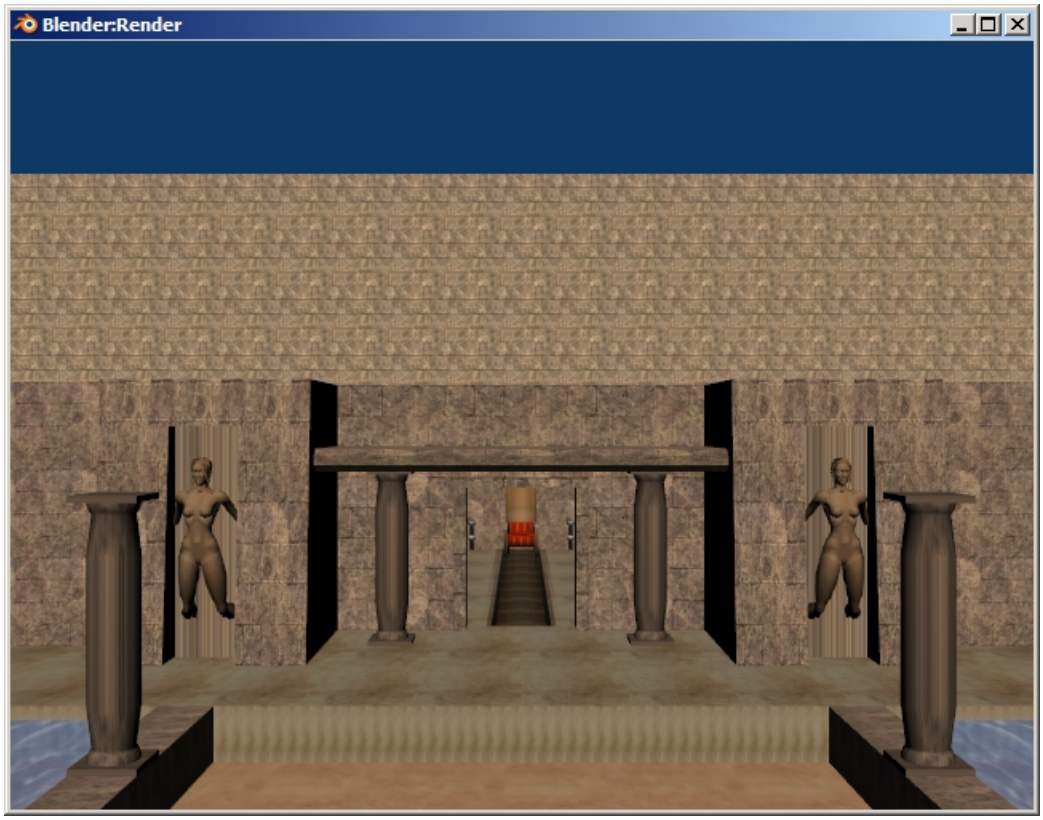


La muraille

Le temple



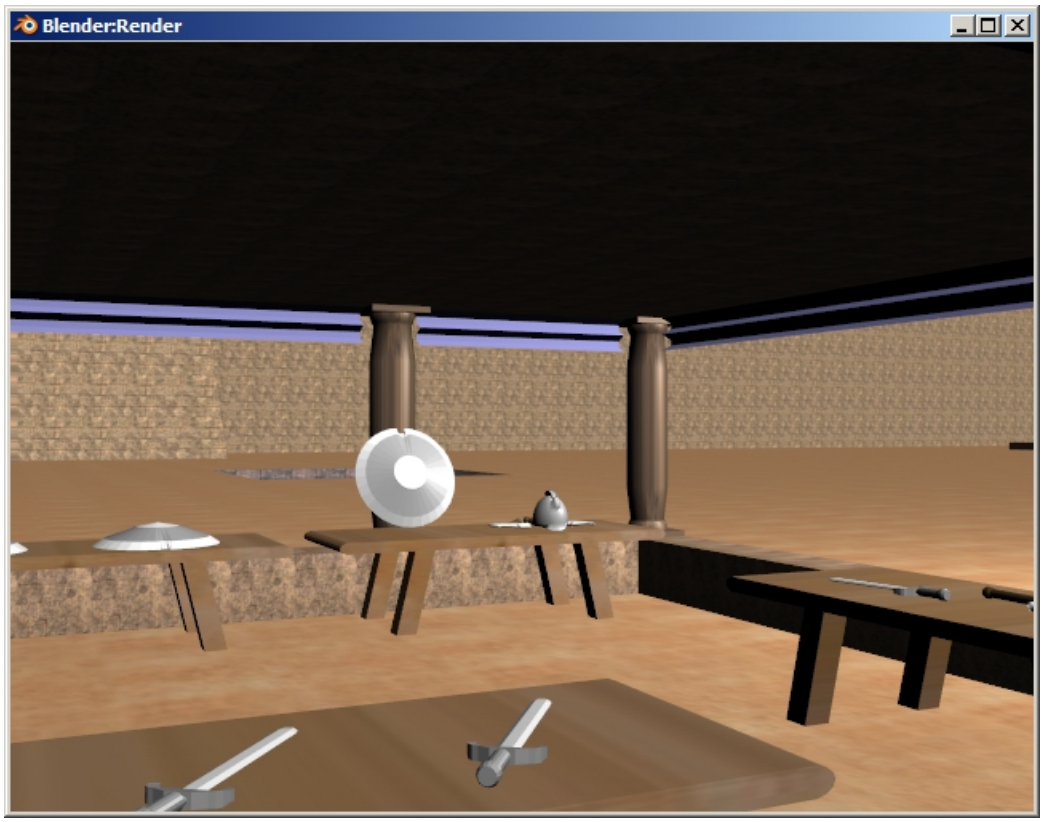
Blender:Render


Blender:Render

Temple vu de l'intérieur

Un marchand d'armes

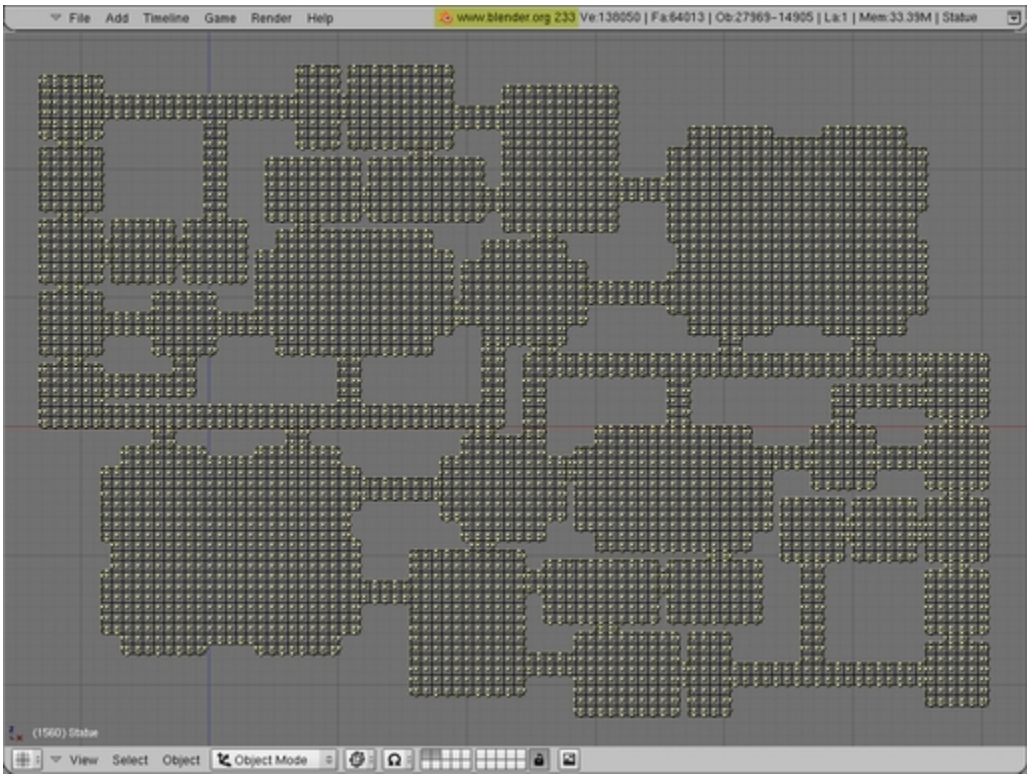
Blender:Render


Blender:Render

Entrée du sous-terrain

Le sous-terrain

Plan du sous-terrain (vue de dessus)


File Add Timeline Game Render Help
www.blender.org 233 Ve:138050 | Fa:64013 | Ob:27969-14905 | La:1 | Mem:33.39M | Statue
(1560) Statue
View Select Object
Object Mode

Nous avons une classe Frustum commune que chacun utilisera en fonction des besoins de son algorithme.
Nous avons aussi implémenté un programme de démo de frustum culling en 2D basé sur cette classe, et vu de dessus (cf Wiki).

# Les quadtrees

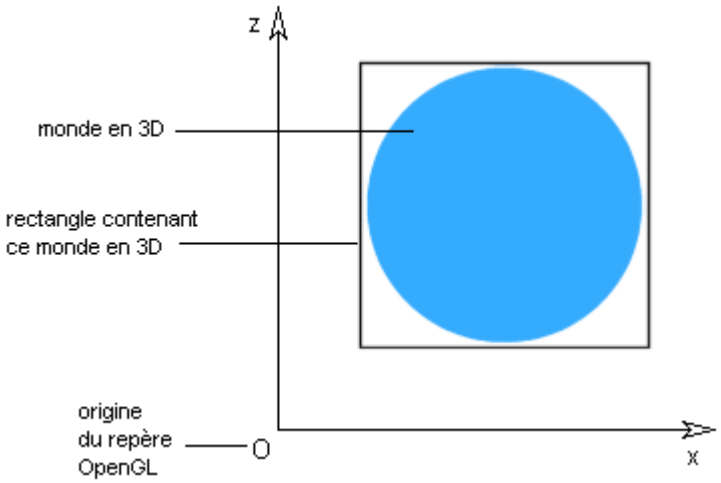
Les quadtrees sont bien plus vieux que les jeux vidéos en trois dimensions. En effet, les quadtrees sont une structure de données inventée en 1974 par Finkel and Bentley et utilisée à la base en traitement et en compression d'images. Nous verrons ici leur utilité dans les applications en 3D.

## Qu'est-ce que c'est ?

Les quadtrees sont des arbres à 4 branches. Ils vont permettre de diviser le monde en 3D en 4 parties.

Prenons l'exemple d'un monde sphérique. Regardons ce monde vu de dessus. Ses dimensions en X et en Z sont comprises dans un rectangle (ou un carré si X = Z).

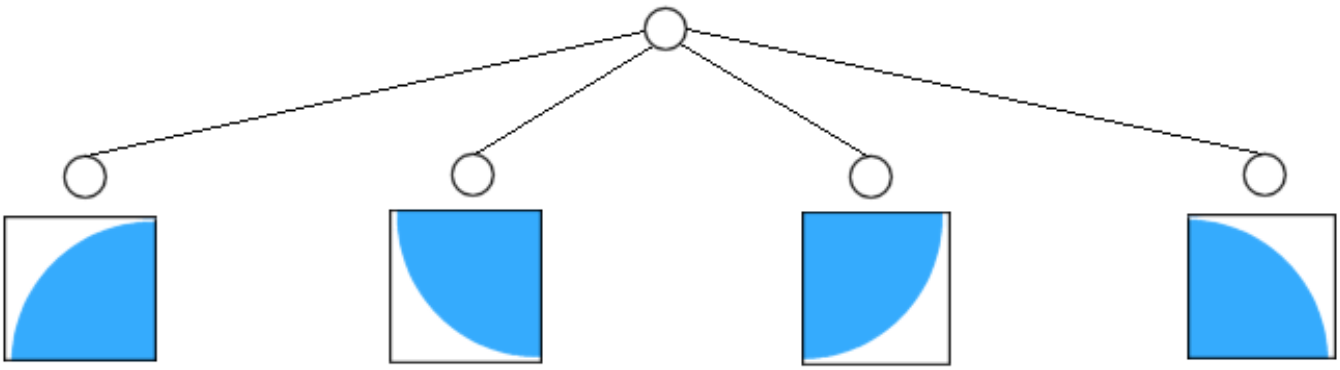
- Illustration :




Ce rectangle va être associé au nœud racine du quadtree.
Il faudra toujours garder à l'esprit qu'un nœud ou une feuille d'un quadtree est associé à un rectangle.

Maintenant, on divise le rectangle en 4 rectangles égaux et de même proportions que celui qui les contient (chacun des 4 rectangles créés a pour taille en X et en Z la moitié de la taille du rectangle qui a été coupé). Chaque fils du nœud racine sera associé à un de ces rectangles.

- Chaque nœud fils avec son rectangle associé :

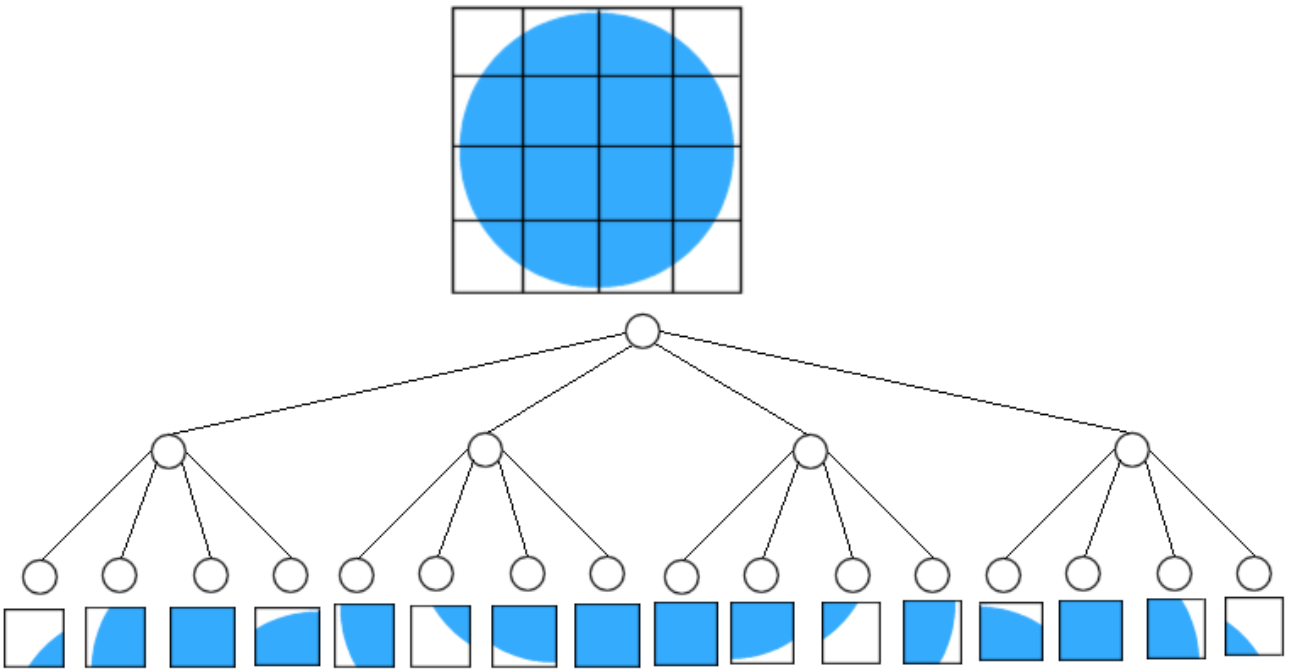


Ci-dessus et dans le programme, on a choisi d'associer :

- le premier fils au coin haut gauche,
- le deuxième fils au coin bas gauche,
- le troisième fils au coin bas droite,
- le quatrième fils au coin haut droite.

Ensuite, chaque rectangle associé à un fils pourra être divisé en 4 et ainsi de suite.
On partitionne donc récursivement l'espace en rectangles.
On appellera **profondeur** le nombre de fois qu'on divisera le monde en rectangles.
Cette profondeur correspond à la hauteur du quadtree. Le rectangle initial correspond à une profondeur de 1.

- Ici en profondeur 3 :



Comme on le voit ci-dessus, chaque feuille du quadtree est associée à une partie du monde, donc chaque feuille connaîtra les polygones situés dans le rectangle correspondant.

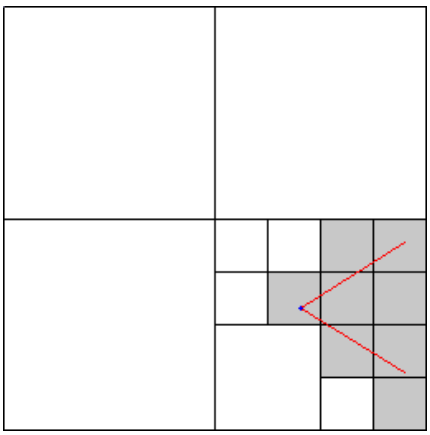
## A quoi ça sert ?

Si on se contente de découper le monde en rectangles, ça ne changera rien à la vitesse d'exécution. Les quadtrees servent à quelque chose si on les utilise avec le frustum culling. On a vu que le frustum culling consistait à éliminer les polygones hors du champ de vision de la caméra, mais aussi qu'il serait trop long de tester si chaque polygone est visible ou pas. Les quadtrees permettent d'éliminer un grand nombre de polygones très rapidement.

Pour ce faire, au lieu de décider si chaque polygone du monde est dans le champ de vision ou non, on va décider si un rectangle contenant des polygones est dans le champ de vision ou non. Ce qui va diminuer de beaucoup le nombre de tests à effectuer.
Explications : considérons un monde en 3D carré, toujours vu de dessus. Imaginons qu'il contient 4096 polygones triangulaires répartis uniformément dans le carré. Tester si chaque polygone est visible reviendrait à faire 4096 * 3 sommets = 12288 tests, pour n'afficher que les polygones visibles et aucun autre.
Maintenant, si on teste si les carrés qui les contiennent sont dans le champ de vision de la caméra, que va-t-il se passer ? On teste d'abord si le carré encadrant le monde est dans le champ de vision. Si non, ça veut dire que la caméra est hors des limites du monde, le dos tourné vers l'extérieur. Donc on arrête et on affiche aucun polygone. Si oui, on va ensuite regarder les 4 carrés correspondant aux fils de la racine du quadtree. Si ils ne sont pas dans le champ de vision, on ne s'en préoccupe plus. Si ils ont visibles entièrement ou en partie, on va encore les diviser en 4 et considérer leurs fils jusqu'à une certaine profondeur spécifiée, avec 1 <= profondeur <= hauteur du quadtree.

Par exemple sur l'image ci-dessous (tirée du programme de démo de frustum culling en 2D fourni en annexe), on est en profondeur 4. Le champ de vision est représenté par les segments rouges (la caméra peut ici voir à l'infini). Les carrés grisés sont ceux qui sont visibles par la caméra. On affiche donc tous leurs polygones.

- Illustration :

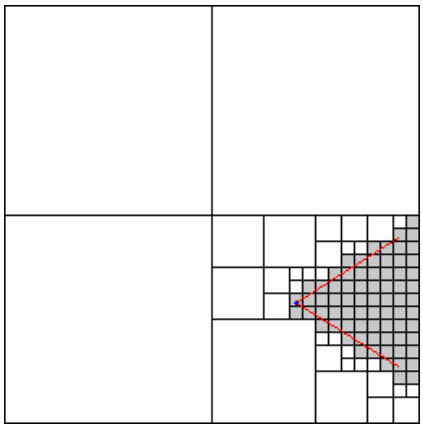


Chacun de ces carrés contient 4096/(4^3) soit 64 polygones. On a 8 carrés en gris donc on affiche 512 polygones sur 4096. On remarque qu'il y a des polygones qui ne sont pas visibles mais qui sont quand même affichés parce qu'un bout du carré dans lequel ils sont contenus est visible. Le nombre de polygones affiché n'est pas minimal. Mais regardons le nombre de tests effectués :

- **4** tests pour le carré initial. Ce carré est visible, donc on le divise en 4 et on regarde chacun de ces fils :
- 4 * 4 = **16** tests pour les carrés de profondeur 2. Mais 3 de ces 4 carrés sont totalement hors du champ de vision donc on ne considère plus que le carré en bas à droite.
- 4 * 4 = **16** tests pour les carrés de profondeur 3. Un des 4 est totalement hors du champ de vision, il en reste donc 3.
- 4 * 4 * 3 = **48** tests pour les carrés de profondeur 4.

Il a donc fallu 4+16+16+48=**84** tests au lieu de 12288 pour savoir quels polygones afficher, donc même si on affiche légèrement plus de polygones qu'en les testant tous, le nombre de tests dérisoires rend la méthode avec les quadtrees beaucoup plus efficace.
De plus, si on descend en profondeur, les carrés vont être de plus en plus petits et le nombre de polygones affichés va se rapprocher du nombre optimal. On pourra dans notre programme observer le nombre de polygones affichés par rapport au nombre total, et la hausse du nombre d'images par seconde que cela entraîne.

- Exemple en profondeur 6 :



Cependant, un parcours récursif est coûteux et quand la profondeur est trop importante les performances se dégradent. Il faut qu'une augmentation de profondeur se traduise par une baisse importante du nombre de polygones affichés pour être efficace.

# Concrètement

On a vu qu'un objet 3D était une structure contenant un tableau de polygones, et qu'on avait une liste d'objets 3D en variable globale. Ici la solution choisie pour représenter les polygones dans le quadtree a été de donner à chaque nœud et chaque feuille un tableau de pointeurs vers les objets 3D du tableau global. On a donc pas de duplication des objets 3D. On a donc pas directement accès aux polygones par le quadtree, il faut passer par l'objet qui les contient, ce qui évite de faire trop de pointeurs et économise donc de la place.

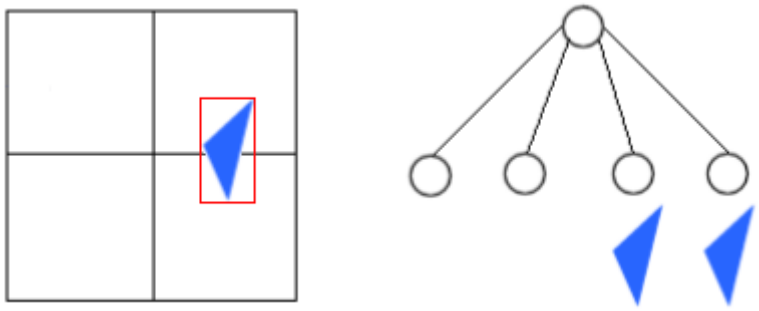
Il a fallu ensuite choisir où placer l'accès aux objets 3D : dans les feuilles ou dans chaque nœud ? C'est la première solution qui a été choisie : on ne remplit que les vector des feuilles. Les nœuds intermédiaires auront accès aux objets récursivement par leurs fils, ce qui évite de copier les pointeurs à chaque niveau.

Dans le moteur 3D de base, à chaque image on parcourait le tableau des objets 3D et pour chacun on affichait tous ses polygones. Maintenant on part de la racine du quadtree et on descend jusqu'à la profondeur voulue, en éliminant les branches correspondant à des rectangles hors du champ de vision. Si on est au niveau des feuilles, on affichera le contenu de chacune d'elles.
Sinon, on descendra récursivement des nœuds où on se trouve jusqu'aux feuilles, pour les afficher.

<u>Voici comment on répartit les objets dans les différentes feuilles :</u>

Pour chaque objet, on considère sa bounding box. Si un des sommets de la bouding box est situé à l'intérieur du rectangle associé à une feuille, on ajoute un pointeur vers l'objet au vector de cette feuille.
Pour aller jusqu'aux feuilles on part du nœud racine, et on descend récursivement en ne considérant que les nœuds fils dont le rectangle contient au moins un sommet de la bouding box.

Illustration sur un exemple simple : on a un quadtree de hauteur 2 et un objet 3D formé d'un seul polygone triangulaire. Le rectangle rouge est sa bouding box, toujours vue de dessus. On voit que 2 feuilles sur les 4 auront dans leur vector le même pointeur sur l'objet.



Ceci nous fait une structure simple de quadtree dans le code :

```
class Quadtree {
public :
   Quadtree *fHG, *fBG, *fBD, *fHD;
   int xmin, zmin, xmax, zmax;
   vector<Object3D *> list_Object;
};
```

- fHG est le fils haut gauche, fBG le fils bas gauche, fBD le fils bas droit et fHD le fils haut droit.
- xmin, zmin sont les coordonnée du sommet en haut à gauche du rectangle associé au quadtree et xmax, zmax celles du sommet en bas à droite.
- list_Object est un vector (tableau dynamique) de pointeurs vers les objets 3D.

Avec cette méthode de nombreux polygones risquent d'être affichés plusieurs fois, c'est pour ça qu'un booléen isDisplayed a été rajouté dans la classe Object3D. Quand on affichera un objet d'une feuille du quadtree, on le marquera comme déjà affiché pour qu'il ne soit pas encore affiché par les autres feuilles. Ceci est possible parce qu'on a des pointeurs vers un tableau d'objets global dans les feuilles. Quand on modifie un Object3D, ça le modifie donc pour toutes les feuilles.

On a parlé du remplissage mais pas de la création des quadtrees.

Avant de remplir l'arbre, il faut le créer. Il y a plusieurs façons de déterminer à quelle hauteur de l'arbre s'arrêter.

- La première est de s'arrêter de créer des fils quand le rectangle associé à un nœud a une taille inférieure à une taille fixée.
- La deuxième pourrait être de s'arrêter quand les rectangles contiennent moins d'un certain nombre de polygones, mais elle ne peut être utilisée que dans des mondes en 3D où les polygones sont répartis périodiquement, sinon on ne connaît pas à l'avance le nombre de polygones contenus dans les rectangles.
- La troisième, celle utilisée ici, est de décider dès le départ de la profondeur.

Notre constructeur de quadtree prend donc en paramètre la profondeur choisie, les coordonnées x et z des coins du rectangle encadrant le monde. Il construira récursivement ses fils en décrémentant la profondeur à chaque niveau, et ce tant qu'elle sera > 0.

# Passons à la 3D

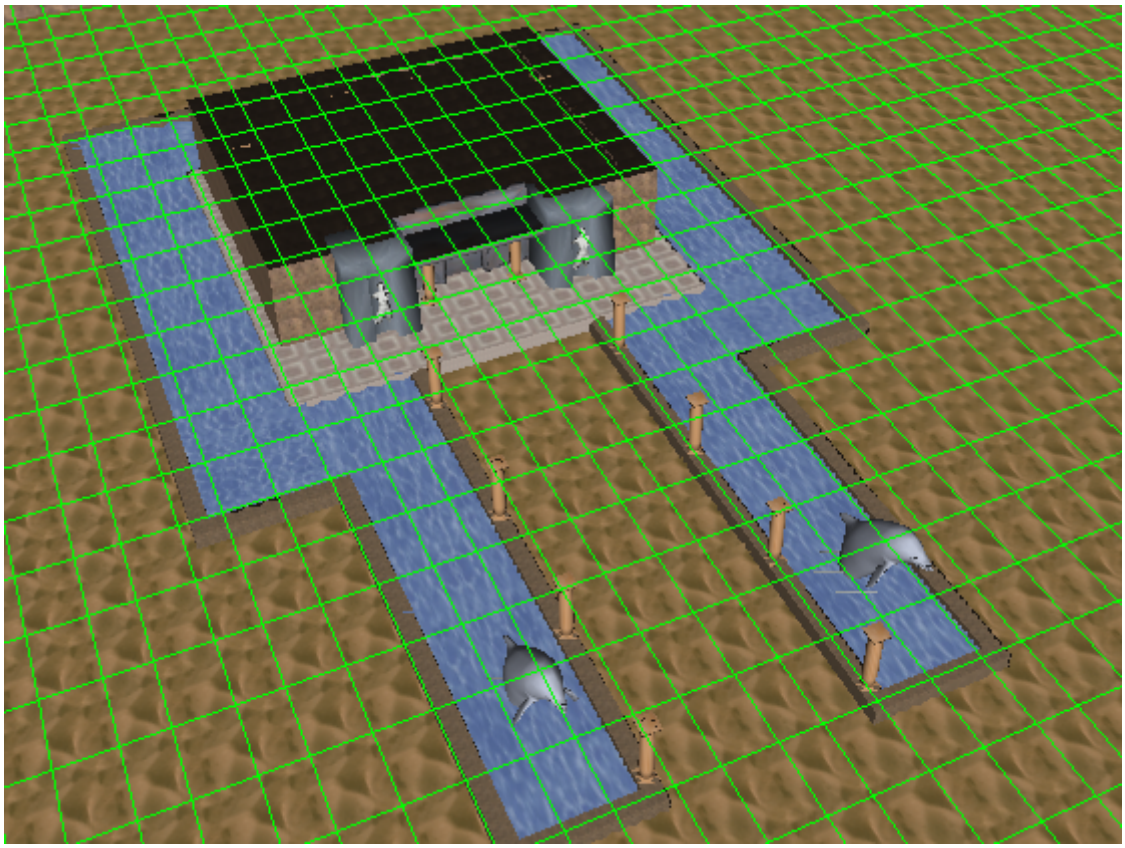
Comment tester si le rectangle (figure en 2D) associé à un nœud ou une feuille du quadtree est à l'intérieur du champ de vision de la caméra, champ défini par des plans dans l'espace (3D) ?
La solution qui vient à l'esprit est simplement de tester la méthode qu'on a vue dans le paragraphe sur le frustum culling avec les 4 sommets du rectangle. Mais quelle

coordonnée en y donner à ces sommets ? On a deux méthodes.
Avant tout, on remarque que l'on n'a pas besoin des 6 plans du frustum, seuls 4 suffisent. En effet avec les quadtrees les polygones sont regroupés dans des rectangles, peu importe leur coordonnée en Y. Donc on a pas besoin de considérer les plans **top** et **bottom** du frustum.

1ère méthode :

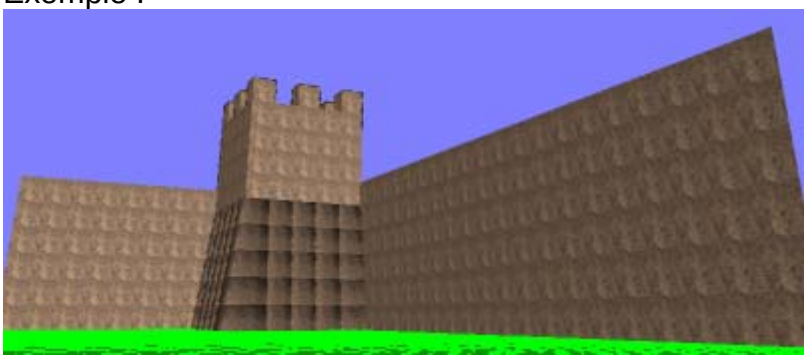
Si on ne met pas de Y, cela revient à dire qu'on considère que les rectangles sont sur le plan d'équation Y = 0. Mais on veut plutôt que ces rectangles correspondent au niveau du sol, qui n'est pas forcément à 0. Le programme permet de monter ou de descendre la coordonnée en Y, et d'afficher le découpage du monde 3D en rectangles.

- Exemple avec Y > hauteur max du temple et profondeur 8 :



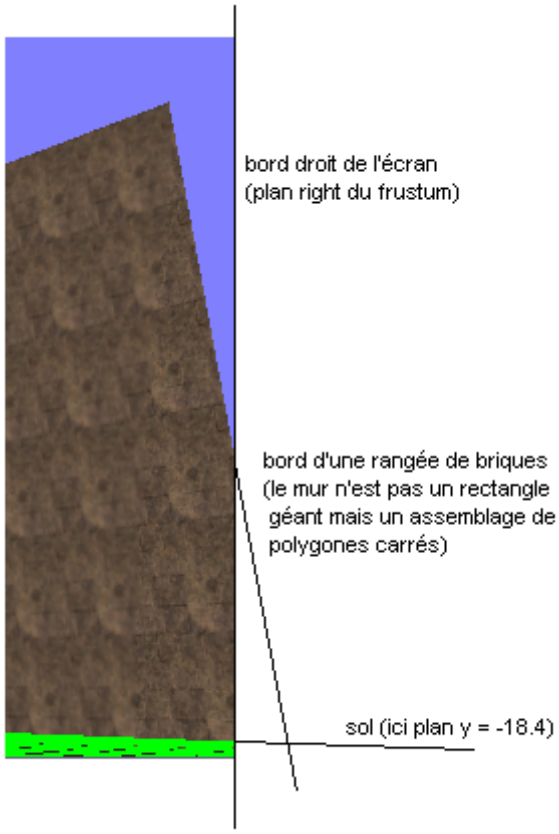
Il y a un problème avec cette méthode : en effet tant que la caméra est à l'horizontale, tout s'affiche correctement mais dès qu'on l'oriente vers le haut ou vers le bas des polygones risquent de ne plus être affichés.

- Exemple :



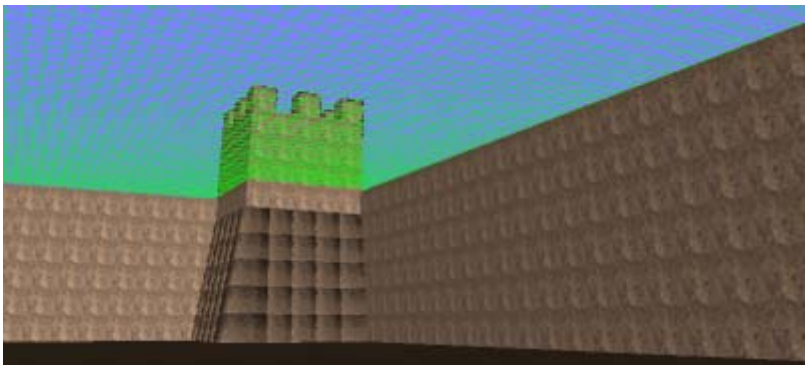
Le problème est que lorsqu'on regarde vers le haut ou vers le bas, la caméra déforme la scène et des lignes verticales apparaissent penchées (comme les murs dans l'image ci-dessus). Si les murs apparaissaient droits, il n'y aurait pas de trous ! Si on fait attention, on remarque **qu'au niveau du sol** (ici le sol est à -19 en y et non 0), le mur qu'on voudrait voir affiché est totalement en dehors du champ de vision, donc les quatre sommets du carré auquel il appartient sont à gauche du plan left du frustum (si le carré est assez petit donc si la profondeur est assez grande), et donc il est normal que ce bout du mur ne soit pas affiché.

- Illustration :


bord droit de l'écran
(plan right du frustum)
bord d'une rangée de briques
(le mur n'est pas un rectangle
géant mais un assemblage de
polygones carrés)
sol (ici plan y = -18.4)

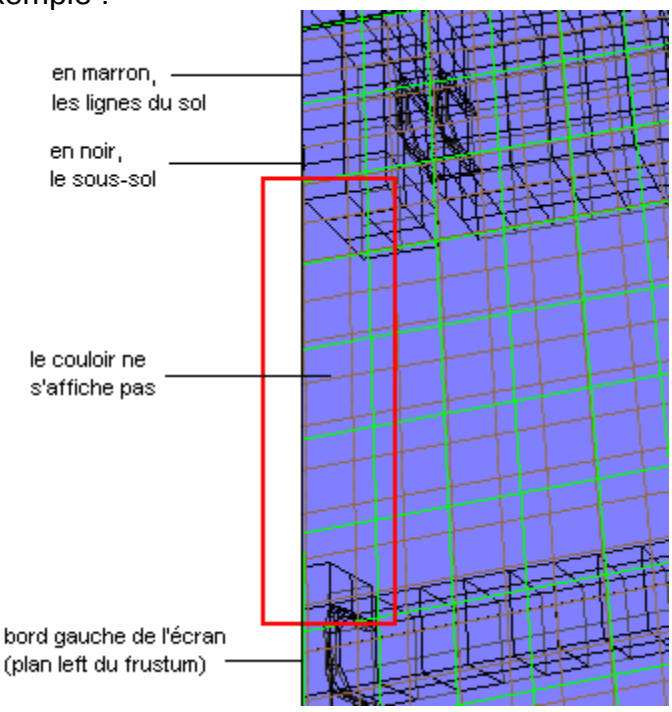
Si on veut voir correctement le mur, il faut tester les équations de plans avec Y = hauteur du mur et non plus Y = 0. Ainsi, à cette hauteur les carrés contenant le mur sont dans le champ de vision :

- ici on teste avec y = +29 :



Seulement maintenant il risque d'y avoir des trous au niveau du sol si on oriente la caméra trop bas, alors qu'avant il n'y en avait pas. Ceci montre que cette méthode n'est pas adaptée pour les programmes permettant d'orienter la caméra n'importe où, puisque quel que soit le plan choisi, il pourra des polygones quelque part.
Toujours le même problème : si on se met en hauteur et qu'on regarde le sol en fixant la hauteur du sol (-19) comme coordonnée y des carrés, le sol s'affichera très bien. Mais si on se met en mode fil de fer et qu'il y a des polygones sous le sol, ceux qui sont sur les côtés risquent de ne pas s'afficher, de la même façon que pour le mur ci-dessus. Ce n'est pas facile de se représenter le problème. C'est une question de perspective, on croit qu'on devrait voir des polygones mais si on les projette sur le plan Y = hauteur, ils sortent du champ de vision.

- exemple :

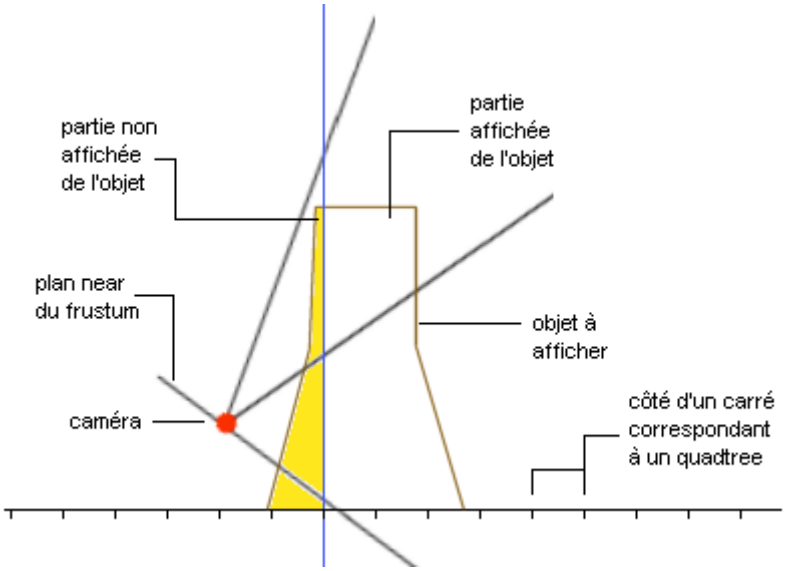


Sur l'image ci-dessus, on voit des zones de l'écran qui ne sont pas affichées (parce qu'on a Y = -19 alors que ces zones sont bien en dessous de -19). Si on abaisse le Y , tout s'affichera mais il y aura encore moins de polygones affichées au dessus du sol quand on regardera en haut.

Ceci montre que cette méthode n'est pas non plus adaptée aux environnements à plusieurs étages.

La déformation de la scène par la caméra n'est pas le seul problème. Imaginons qu'on fixe Y = hauteur du sol pour les rectangles. Si on est dans le temple et qu'on regarde vers le plafond, on va voir le ciel. C'est normal, on teste si les carrés sous nos pieds sont au dessus de notre tête. Bien sur le test est négatif donc on affiche pas tous les rectangles qui ont leurs 4 sommets derrière le plan **near** du champ de vision de la caméra.

- Illustration :

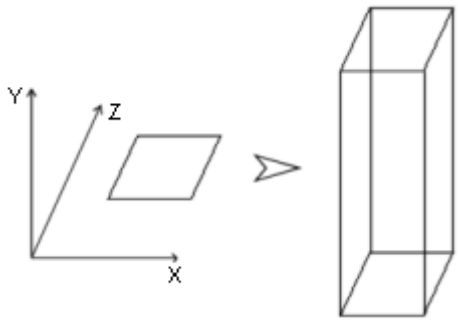



En fait, cette méthode n'est satisfaisante qu'avec des applications où les polygones sont tous posés sur un seul plan et où on ne peut pas orienter librement la caméra. Typiquement des jeux de courses de voitures, ou de stratégie avec vue de dessus par exemple.
Mais pour les jeux de type FPS, ça ne convient pas et il faut trouver une autre solution.

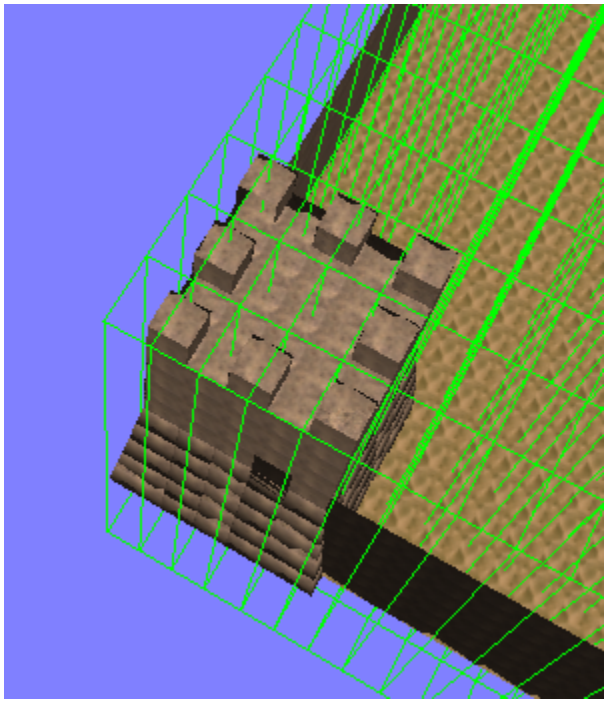
<u>2ème méthode :</u>

La 2e méthode est plus lente mais elle permet d'afficher dans tous les cas tous les polygones visibles.
Il faut pour cela ne plus considérer des rectangles sur un plan horizontal mais les élever dans l'espace en colonnes. Le haut de ces rectangles sera la coordonnée Y maximale du monde en 3D et le bas le Y minimal.

- Exemple :




- Illustration avec le moteur 3D :



Le programme permet de passer d'une méthode à l'autre. Voir le mode d'emploi fourni en annexe sur le Wiki pour connaître les fonctionnalités associées à chaque touche.

Pour savoir si un rectangle est dans le champ de vision, on teste deux fois si ses sommets sont dans le champ de vision, une fois avec Y = hauteur maximale du monde et une fois avec hauteur minimale du monde. En fait on teste avec les 8 sommets de la colonne. Ca fait deux fois plus de tests qu'avec la première méthode mais on est sur que tous les polygones visibles sont affichés.

Par contre, on risque parfois d'afficher beaucoup de rectangles inutiles parce qu'on un petit bout de leur colonne est visible, même s'il n'y a pas de polygones dans la partie qu'on voit, ce qui fait que les performances sont moins bonnes qu'avec la 1ère méthode.

Et quand quand on affiche un rectangle, on affiche tous ses polygones, quelle que soit leur hauteur, donc par exemple si on est au dessus du sol on va quand même afficher tous les polygones du sous-sol qui appartiennent aux rectangles du champ de vision, ce qui diminue les performances quand il y a des étages. On verra que les octrees répondent à ce problème.

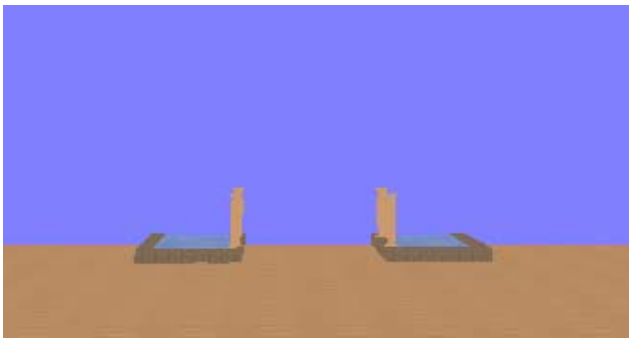
# Clipping à l'horizon

On remarque en testant le programme que les performances varient beaucoup selon l'endroit du monde où l'on regarde. Par exemple, si on se met en hauteur et qu'on regarde en bas, on voit tous les polygones du monde et le nombre d'images par seconde est aussi bas que s'il n'y avait pas de quadtrees. Par contre, si on se place dans un coin et qu'on oriente la caméra vers une tour, la rapidité sera optimale. Parfois les terrains sont immenses et on ne peut pas afficher les polygones jusqu'à l'infini. De plus, il y a souvent de fortes chances pour que les polygones lointains soient cachés par d'autres polygones plus proches, et donc pas ou très peu visibles. La solution utilisée dans de nombreux jeux vidéos est de clipper les polygones à l'horizon. De la même manière qu'on ne souhaite pas afficher les polygones qui étaient à gauche ou à droite du champ de vision de la caméra, on n'affichera pas ceux qui sont plus loin que le plan **far** du frustum.

Au lancement du programme, le plan far est à une distance de 1000 unités Open GL de la caméra, ce qui fait que quelque soit l'endroit à l'intérieur des murailles où l'on se place, si on est au niveau du sol on pourra quand même voir les polygones les plus éloignés.
Le programme permet d'avancer ou de reculer ce plan far par rapport à la caméra.

Le problème quand on n'affiche pas les polygones qui sont derrière le plan far est que ça crée une coupure très visible qui n'est pas agréable pour l'utilisateur. Pour résoudre ce problème, notre programme, comme un certain nombre de jeux, utilise le brouillard (fog). On rajoute avec Open GL un effet de brouillard au niveau de la cassure, qui la masque progressivement.

- La même image sans et avec brouillard :



Le fait d'avancer le plan far et de mettre du brouillard est très pratique dans les couloirs de notre sous-sol par exemple, où l'on n'a pas besoin de voir très loin, et où on peut donc se promener avec beaucoup d'images par seconde.

# Optimisations possibles

## Level Of Details

Mais le clipping à l'horizon est rendu obsolète par les capacités des cartes graphiques actuelles.
Nicolas Peri nous a dit qu'aujourd'hui, une méthode très utilisée est le **Level Of Details (LOD)**.
Nous ne l'avons pas implémenté mais voici des explications :

On calcule tout d'abord plusieurs représentations du monde en 3D, plus ou moins détaillées.
Au lieu de ne pas du tout afficher les objets trop lointains et de le masquer par du brouillard, on affiche une simplification des objets. Par exemple les statues de notre monde contiennent chacune plusieurs milliers de polygones. Cela ne sert à rien d'en afficher autant quand on est à l'autre bout du monde, car on ne peut les distinguer. On ne verrait pas la différence avec un assemblage d'une dizaine de polygones constituant une statue avec une forme très simplifiée.

C'est ce que permet le LOD. Pour chaque feuille du quadtree visible, on affiche une représentation ou une autre des objets qu'elle contient en fonction de la distance qui la sépare de la caméra.
Les représentations simplifiées sont calculées en fusionnant des polygones voisins. Un programme comme VizUp fait ça très bien. Nous l'avons testé avec une version VRML de l'objet vache du cours de synthèse d'images.

- Voici des images :

- La première est la vache normale, avec tous ses polygones.
- La deuxième vache est simplifiée à 60% : elle contient seulement 40% du nombre des polygones de la première. De près on ne voit pas bien les différences avec la première alors de loin ça passerait totalement inaperçu.
- La troisième elle est simplifiée à 90% par rapport à la première.

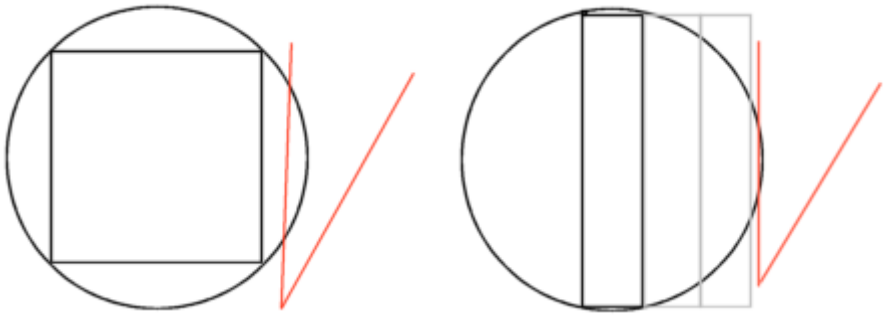
Dans Quake3 par exemple, les personnages et les items (armes, bonus...) sont représentés en trois niveaux de détails.
Le Level Of Details, combiné aux quadtrees, donne de très bons résultats.

## Optimisation du frustum culling

Pour savoir si un rectangle était visible ou non, on devait tester si chacun de ses 4 (ou 8 sommets selon le mode de test) était à l'intérieur du champ de vision ou pas.
Il est possible de diminuer le nombre de tests en les remplaçant par un test d'intesection d'une sphère avec le frustum. Cette sphère a pour centre le centre du rectangle à tester et passe par tous les points du rectangle. Son rayon est la distance de son centre à n'importe quel point.

Pour savoir si un rectangle est dans le champ de vision, on calcule la distance entre son centre et les plans du frustum, au lieu de calculer la distance entre chaque sommet du rectangle et les plans du frustum. C'est une approximation, il peut y avoir des rectangles affichés alors qu'ils sont non visibles, mais ça réduit le nombre de tests à 1 au lieu de 4 pour chaque plan du frustum. Cependant, cette approximation est acceptable seulement quand les rectangles ressemblent à des carrés, sinon, on va trop afficher trop de rectangles inutiles.

- Illustration : avec des carrés, pour chaque carré on peut au pire en afficher un seul qu'on ne voit pas. Avec des rectangles quelconques il peut y en avoir plus.



Dans notre programme avec les quadtrees, on a un rectangle initial de côtés (X, Z). Pour pouvoir utiliser cette méthode il vaut donc mieux avoir un carré initial de côté max(X, Z) quitte à avoir des carrés vides. Cette optimisation n'est pas possible quand on considère les rectangles comme des colonnes (voir ci-dessus la 2e méthode de test de frustum), dont la taille en Y est bien supérieure à celle en X et Z.

## Quadtrees dynamiques

Parfois, les mondes en 3D sont tellement grands (plusieurs centaines de Mo voire plusieurs Go) qu'on ne peut pas les charger entièrement en mémoire.
On ne peut donc pas construire un quadtree contenant tous les polygones dès le début de l'exécution. Dans ces cas là, le programme lit sur le disque les informations

sur le monde en 3D en fonction de la position de la caméra dans ce monde, et modifie le quadtree en fonction de ce qu'il a lu.

# Conclusion

Les quadtrees sont une structure de données assez simple qui permet de découper l'espace en rectangles. L'algorithme ne comporte pas de grosses difficultés à implémenter mais n'est vraiment efficace que dans des cas précis. Par exemple, il est optimal quand la caméra a une orientation fixe et qu'on est dans un environnement extérieur, et sans étages. Dans notre environnement, ce n'est donc pas l'algorithme le plus efficace. En fait, nous nous sommes aperçus que les quadtrees ne sont pas vraiment adaptés aux jeux 3D de type FPS où l'orientation de la caméra est libre et les niveaux très variés.

# Les Octrees

## Introduction

Un octree est une manière de subdiviser l'espace 3D.
Le premier objectif des octrees est de réduire le nombre de comparaisons requis afin de déterminer les polygones de notre monde que l'on va traiter, qu'il s'agisse d'affichage, de tests de visibilité, de tests de collision ou bien de niveau de détails.
Les octrees engendrent une réduction significative du temps requis pour trier les polygones dans un monde et les afficher et ils s'avèrent être le plus optimal pour des jeux présentant de grandes variations, relativement à un axe (par exemple Flight simulator).

Il est important de comprendre que l'application de l'algorithme des octrees n'est qu'une étape dans le processus d'élimination des surfaces et ainsi son unique utilisation sera inutile et engendrera une perte de performance considérable.
En effet celui-ci doit être couplé au processus d'élimination en fonction du point de vue (Frustum culling).

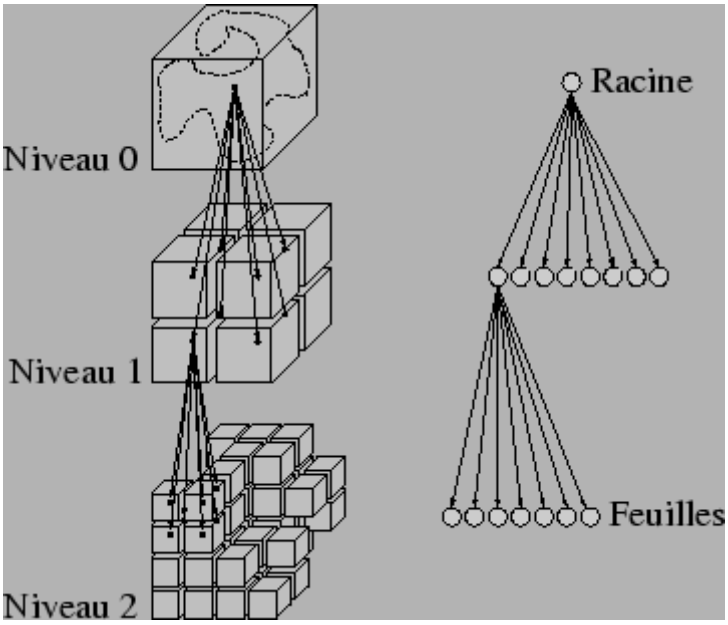
Il existe deux formes d'OSP (Octree Space Partionning):

1. les OSP travaillant sur les objets de la scéne.
2. les OSP travaillant sur le volume de l'espace que forme notre monde.

La première forme d'OSP stocke ses données dans un "n-tree" c'est à dire un arbre dont les nœuds parents ont un nombre variés de fils, de zéro à n.
Le nœud racine de l'arbre contient tous les objets de la scène. Plus les nœuds s'éloignent du nœud racine moins ils contiennent d'objet, jusqu'à ce qu'on ait une feuille qui contienne seulement un objet ou une surface.
L'objectif de cet algorithme est de gérer et trier les informations concernant les objets dans la scène relativement aux autres, en minimisant le temps nécessaires pour afficher les surfaces dans un ordre correct : de devant en arrière ou l'inverse.
La deuxième forme d'OSP est idéale pour les applications dans lesquelles la plupart de l'espace est occupé par des objets, et dans lesquelles les objets ont approximativement la même taille. L'objectif de cet algorithme est de gérer les informations de la scène en utilisant une structure "octal-tree" (dans laquelle chaque nœud parent possède huit fils) pour stocker les données.
Il semble effectivement bien adapté pour traiter notre monde c'est pourquoi c'est ce deuxième cas que nous avons implémenté et que nous allons étudier.

## Structure et méthode de création

Un octree n'est rien d'autre qu'une structure de donnée.
Voici un schéma qui met bien en évidence la relation entre le découpage de l'espace et la structure d'arbre :

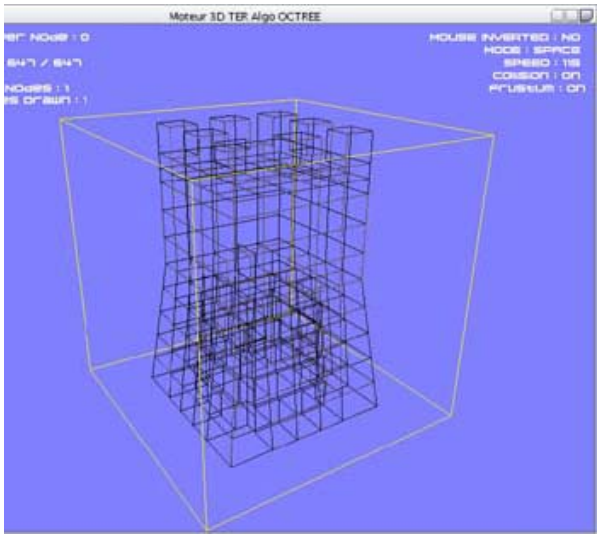


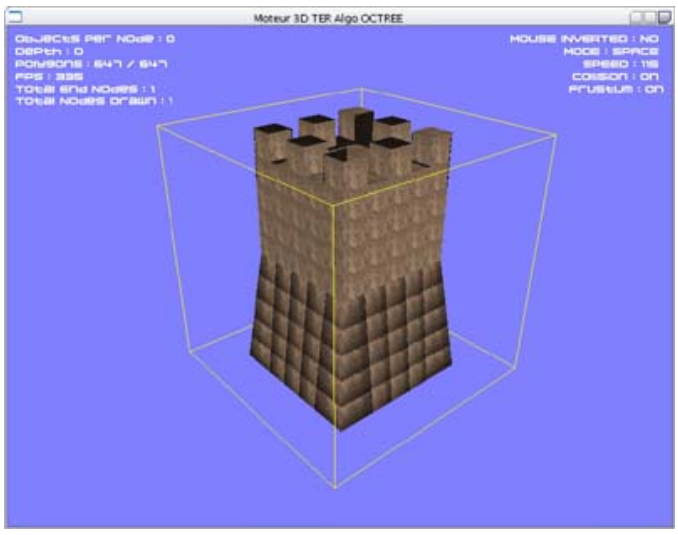

Nous allons expliquer la méthode de création étape par étape.
Pour construire les structures dont on a besoin, l'idée est de procéder de manière récursive. On va simultanément créer une suite de découpages imbriqués et une structure d'arbre associée.

- ## Etape 1 / Niveau 0 dans l'arbre

Pour créer l'octree initial, on cherche à obtenir la dimension maximum de notre scène (notre monde), puis on ajuste les dimensions pour obtenir un cube.
Il s'agit en fait du cube englobant de la scène entière et on affecte ainsi ce cube au nœud racine de notre arbre.

Ce nœud racine contient donc à ce niveau tous les polygones de la scène.
Supposons que la scène soit cette tour, il s'agit d'une des tours de notre monde.
Pour chaque étape nous verrons pour une même profondeur les deux modes de rendu car dépassé une certaine profondeur, le mode fil de fer n'est plus très explicite.



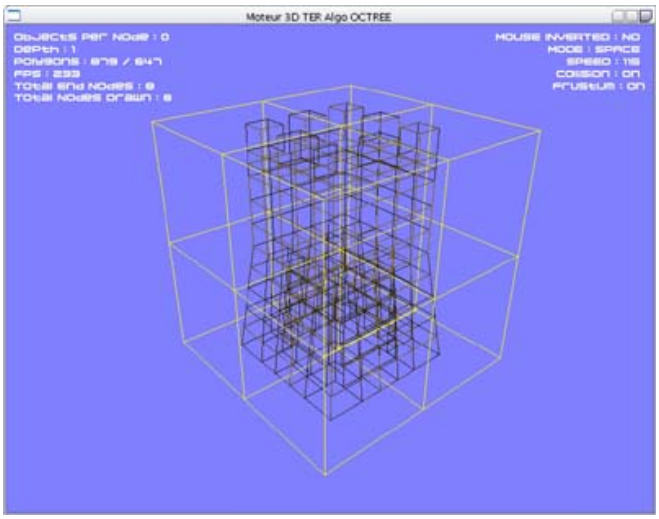



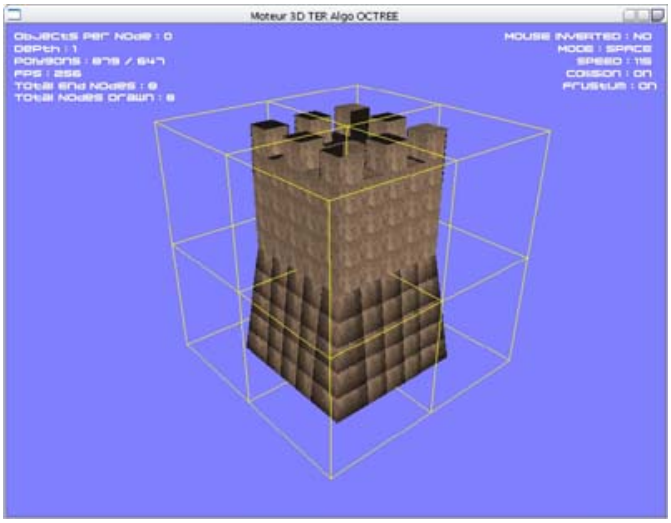
Nous pouvons donc observer malgré que ce ne soit pas trés lisible :

- profondeur est de 0 ("Depth 0" sur l'image)
- l'arbre contient un nœud ("Total ends nodes 1") car un cube a été créé
- un nœud est effectivement affiché ("Total nodes drawn 1") puisqu'il est dans notre champ de vision.

## Etape 2 / Niveau 1 dans l'arbre

Ensuite, on subdivise le cube englobant en 8 cubes plus petits à l'intérieur du cube créé précédemment; on obtient donc 4 cubes en haut et 4 cubes en bas.
On ne garde que les cubes ayant une intersection avec notre monde, en d'autres mots les cubes contenant au moins une surface.



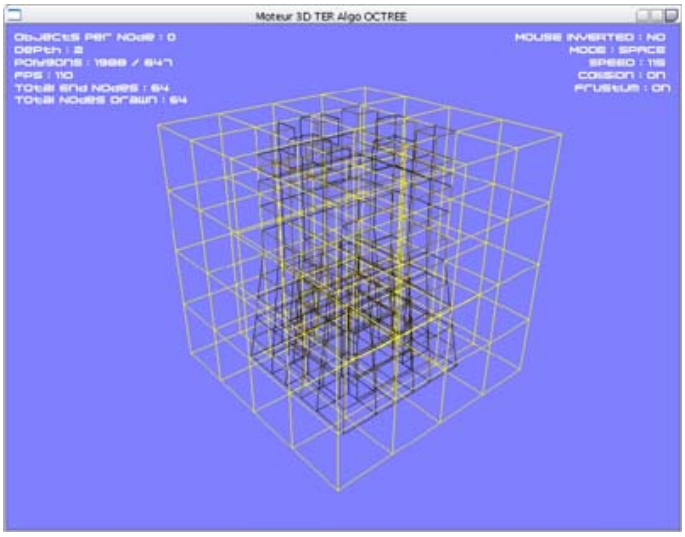



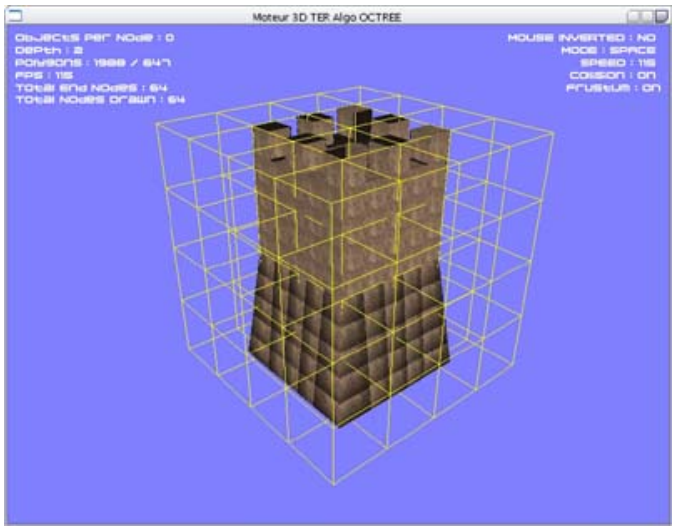

On observe à présent :

- profondeur est de 1 ("Depth 1")
- l'arbre contient 8 nœuds au total en fait ce sont 8 feuilles ("Total ends nodes 8")
- 8 nœuds sont effectivement affichés ("Total nodes drawn 8") puisqu'ils sont tous dans notre champ de vision.

## Etape 3 / Niveau 2 dans l'arbre

On répète ce processus de subdivision pour obtenir le niveau 2 de l'arbre. A chaque nouveau niveau créé, la taille des cubes est divisée par deux par rapport au niveau précédent.





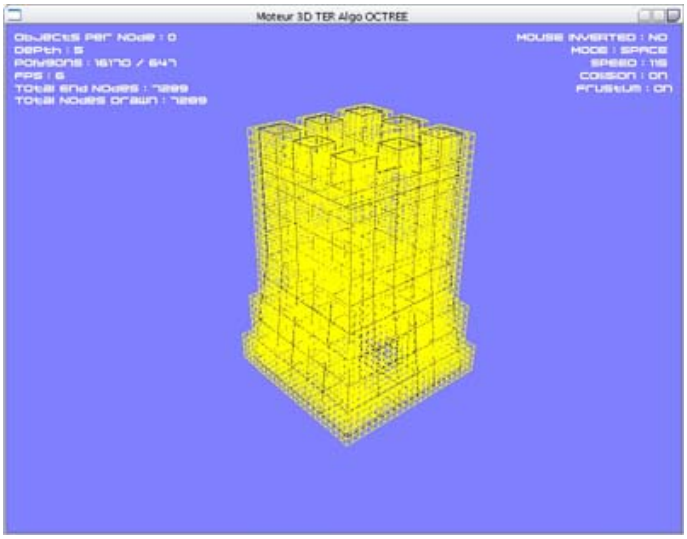

On observe maintenant :

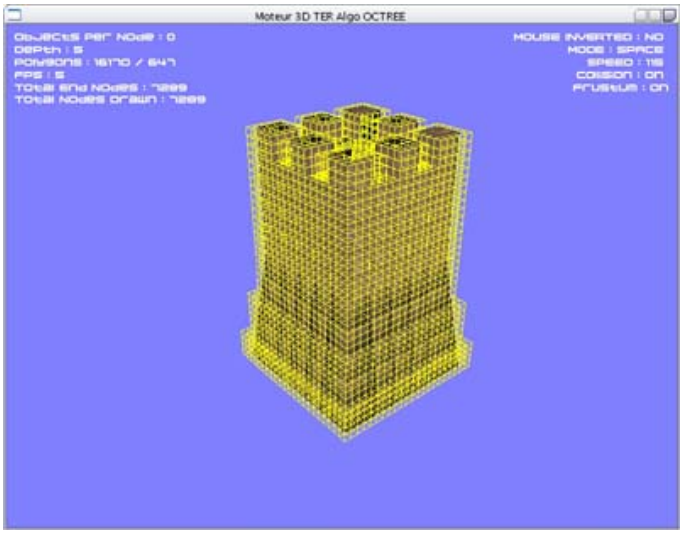
- profondeur est de 2 ("Depth 2")
- l'arbre contient 64 noeuds au total ("Total ends nodes 64")
- 64 nœuds sont effectivement affichés ("Total nodes drawn 64") puisqu'ils sont tous dans notre champ de vision.

on saute quelques étapes ...

- ## Etape 6 / Niveau 5 dans l'arbre

Il est très intéressant d'observer que plus le niveau de profondeur est bas, plus l'agencement des cubes se rapproche de la forme de l'objet.
Ce phénomène est du au fait qu'on ne garde pas les cubes vides et seuls les cubes contenant des surfaces sont créés et affichés.





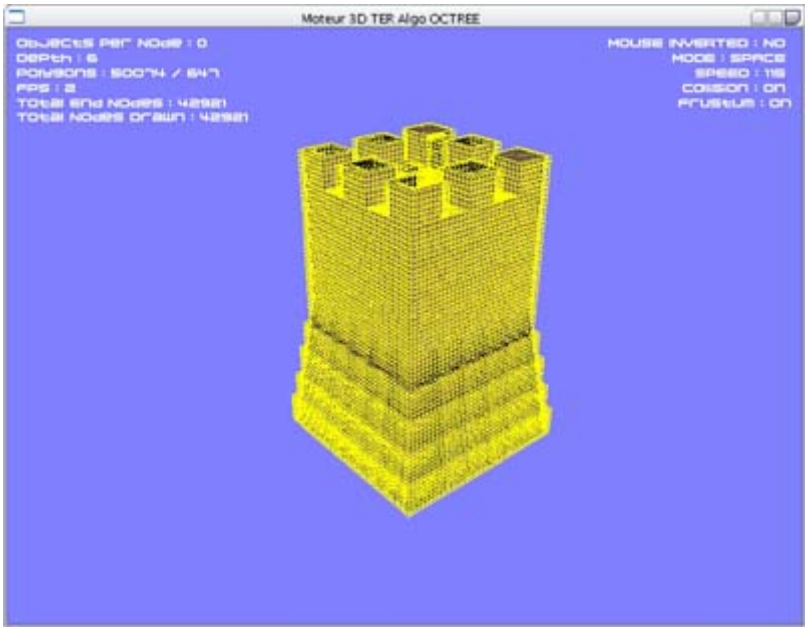
On observe maintenant :

- profondeur est de 5 ("Depth 5")
- l'arbre contient 7829 nœuds au total ("Total ends nodes 7829")
- 7829 nœuds sont effectivement affichés ("Total nodes drawn 7829") puisqu'ils sont tous dans notre champ de vision.

pour finir

- ## Etape 7 / Niveau 6 dans l'arbre

Nous voici donc à une profondeur à laquelle on ne distingue plus du tout l'objet en mode fil de fer donc voici uniquement l'image en mode texturé.




L'arbre se compose de 42921 nœuds terminaux c'est à dire de feuilles contenant des surfaces; le même nombre de nœuds est affiché puisqu'ils sont tous dans le champs de vision de la camera (dans le frustum).

- **Cas d'arrêt**

Le processus de subdivision étant récursif, on ne va pas continuer de subdiviser à l'infini.

Il y a en effet différentes manières de mettre fin à l'itération :

  - quand on a construit un nombre de niveaux fixé par avance
  - la taille des cubes atteint un certain seuil
  - le nombre de polygones maximum (fixé à l'avance) que doit contenir un cube a été dépassé.
  - le nombre de nœud de maximum fixé a été dépassé

Dans les exemples ci-dessus, la profondeur est un paramètre réglables par l'utilisateur donc le cas d'arrêt correspond au niveaux de profondeur.

- **Calcul du nombre d'octree**

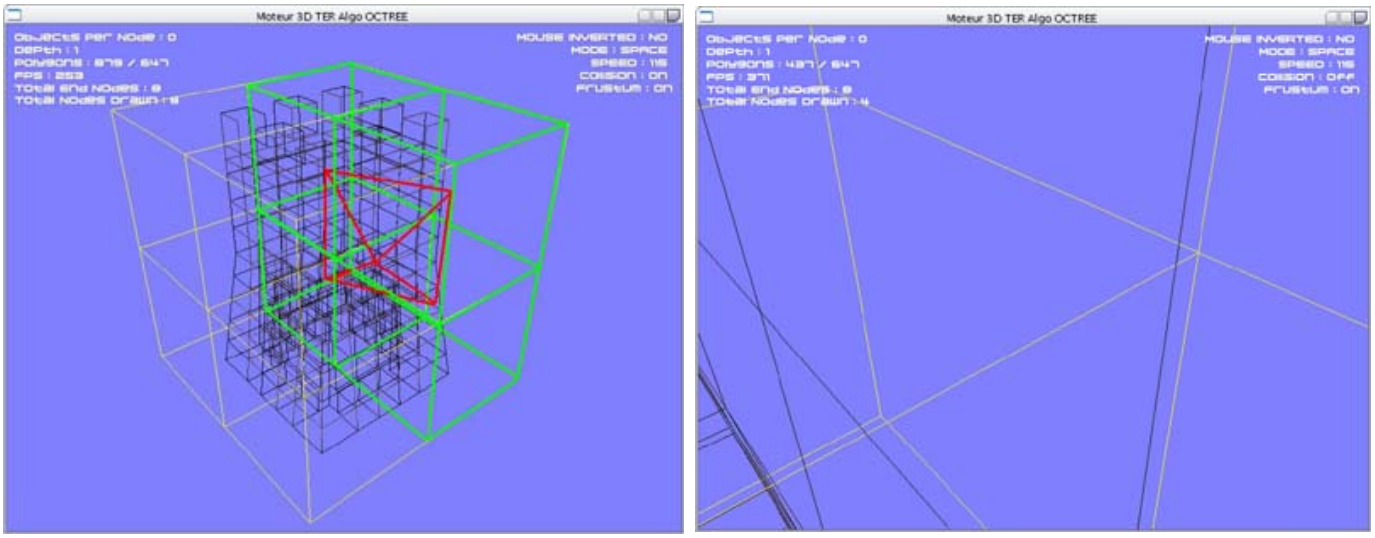
| Profondeur | Octal | Taille de l'octree | Nb total d'octree |
|---|---|---|---|
| 0 | 8'0 | 1024 | 1 |
| 1 | 8'0+8'1 | 512 | 9 |
| 2 | 8'0+8'1+8'2 | 128 | 73 |
| 3 | 8'0+8'1+8'2+8'3 | 64 | 585 |
| 4 | 8'0+8'1+8'2+8'3+8'4 | 32 | 4681 |
| 5 | 8'0+8'1+8'2+8'3+8'4+8'5 | 16 | 37449 |
| 6 | ... | 8 | 299593 |

# Utilisation

Une fois que le processus de création de l'arbre est terminé, on possède une structure de données décrivant un volume lié à l'espace occupé par notre scène, organisé de manière hiérarchique.

Comme nous l'avons vu au début, le partionnement d'espace seul est inefficace, et ainsi il doit être au moins couplé au frustum culling.
Le but du frustum culling est d'essayer de trouver une méthode pour n'afficher que les polygones vus par le frustum de vue. Le frustum de vue peut être considéré comme l'angle sous lequel est vue une scène.
On peut l assimiler au champ de vision chez l humain par exemple.
Le point fort du frustum culling intervient lorsqu'on travaille avec une structure hiérarchique.
Notre monde étant découpé, partitionné dans une structure d'arbre.
Si l'on teste un nœud de haut niveau, alors on a pas besoin de tester les nœuds de niveau inférieur. On en déduit par cela que l'on ne va pas faire le test du frustum sur tous les objets du monde, mais on va les ranger de manière hiérarchique.
Considérons que la scène est constituée de 100 objets, avec seulement une de visible et qu'on les range de manière hiérarchique. Normalement avec une méthode linéaire, on va simplement tester chaque objet et regarder si oui ou non il est visible ou pas. Il en résulte donc 100 tests.
Maintenant considérons le cas binaire. Avec une première vérification on peut réduire le nombre de 50, avec la seconde de 25, ensuite de 13 ensuite de sept puis de quatre puis deux puis une. Ce qui fait six vérifications au total, ce qui est beaucoup plus rapide que 100.
En fait la méthode linéaire doit effectuer N itérations la ou la méthode binaire en effectue seulement log N (log base 2).
Ainsi chaque fils occupent un morceau du volume de l'octree parent.
Lorsqu'on sait que le nœud parent n'est pas visible, on peut affirmer que les nœuds fils ne le seront pas non plus.
En gérant le monde de cette manière on peut rapidement cacher et afficher une très grande partie du monde avec seulement quelques tests.

Voici un exemple d'utilisation du frustum culling :




En rouge sur l'image de gauche le frustum de vue et a droite la vue réelle de la caméra à partir du point rouge.
Si l'on active le frustum culling seul les 4 cubes dessiner en vers sur l'image de gauche sont réellement dessiner sur l'image de droite. Ceci est d'ailleurs confirmer par le nombre de nœuds affichés ("Total nodes drawn 4") comparé au nombre total de nœuds créés ("Total end nodes 8").

## Optimisations possibles :

La première optimisation est assez triviale. Si l'on étudie les deux méthodes vues précédemment d'intersection avec le frustum, il est incontestable que le test d'intersection de la sphère avec le frustum est beaucoup plus rapide que le test d'intersection du cube.
Ceci veut dire qu'il serait plus convenable d'effectuer d'abord le test de la sphère avant de faire celui du cube.
La boite englobante de l'objet est dans certain cas mieux qu'une sphère englobante du fait qu'elle s'adapte beaucoup mieux que la sphère.
En fait chaque objet devrais contenir une bounding box et une bounding sphère.
De ce fait on pourrait rejeter certain objet directement avec le test de la sphère et faire le test avec la bounding box uniquement si le test de la sphère réussi.
La seconde optimisation intervient dans le parcours de l'architecture. Dans notre cas, la méthode générale est de commencer la recherche à partir du nœud racine et de vérifier s'il est visible. Si c'est le cas, on va vérifier chacun des 8 nœuds fils sinon on arrête la recherche.

Cette méthode est récursive et s'applique donc sur tous les nœuds du cube. Une fois qu'on arrive aux nœuds terminaux de l'arbre, s'ils s'avèrent visibles alors on affiche leur contenu.
Maintenant considérons un nœud non terminal, une feuille, si ce nœud est complètement visible, quel est l'intérêt de vérifier la visibilité des nœuds fils, on va utiliser à tord des cycles CPU pour calculer des choses que l'on sait déjà. Dans ce cas on affiche le nœud ainsi que tous ses descendants sans tester les nœuds.

Par contre le seul cas ou on doit tester les nœuds fils est quand le nœud racine intersecte avec le frustum.
La troisième optimisation est de ne pas vérifier l'intersection si la camera est complètement à l'intérieur.
Dans ce cas la on sait que le volume intersecte avec le frustum. Ensuite on procède comme on l'a expliqué ci dessus.

# Implémentation

# Stockage des Octree / Conception de l'arbre

- ## Introduction

Une fois les données du fichier mises en mémoire, il faut maintenant créer les octree comme définis en théorie.

Pour cela, on a créé un module **Octree** ainsi qu'un module **VisualOctree**.
Au cours de notre implémentation, nous avons suggéré, bien que cela ne soit pas rentable au niveau performance, de clipper les polygones lorsque ceux-ci se trouvent à cheval sur deux cubes.

Cette notion de clipping a constitué temporellement une entrave dans notre progression, c'est pourquoi une section consacrée à celui ci a été votée afin d'expliquer les problèmes rencontrés et pourquoi son implémentation n'est pas arrivé à terme.

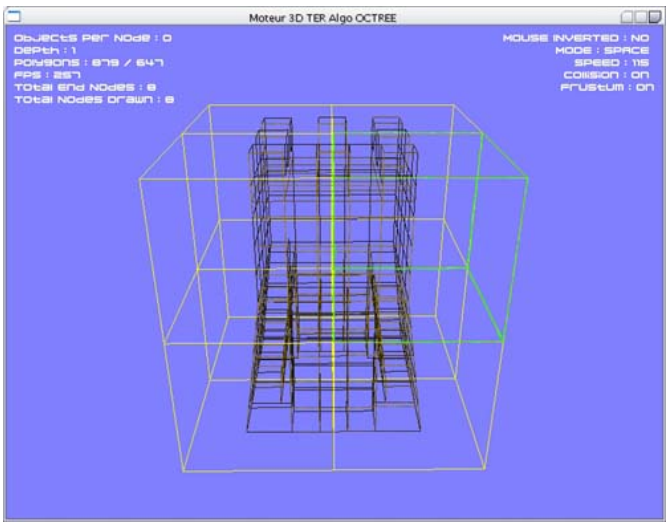
- ## Le problème du clipping de polygones

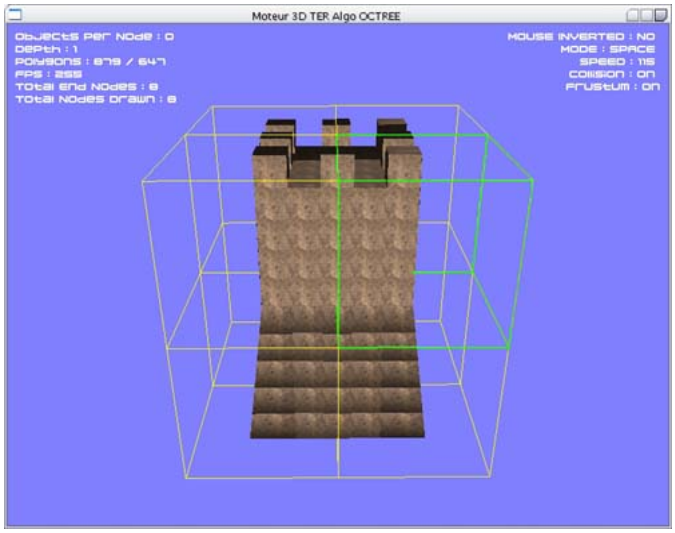
    - **Mise en situation**

"Clipping" signifie coupure. Alors pourquoi souhaite-t-on couper un polygone ?
De manière général on coupe un polygone au niveau du champs de vision, effectivement ça ne sert a rien de balayer un polygone dans sa partie externe du champs de vision.

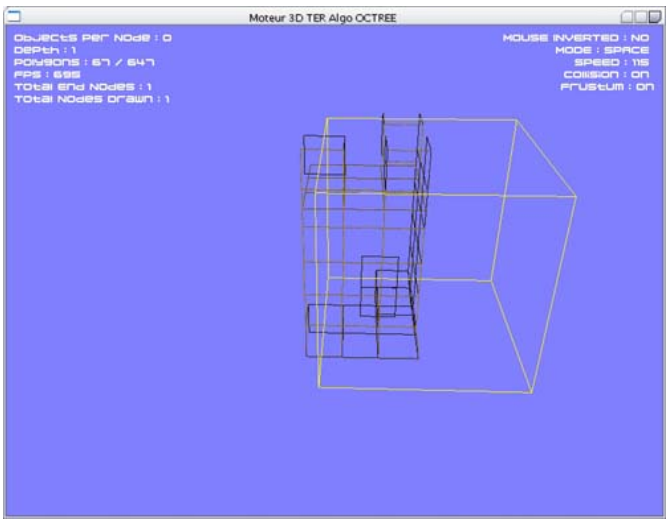
Ici ce n'est pas de ça qu'il s'agit. On souhaite couper les polygones qui chevauchent des cubes lors du découpage du monde en cube.

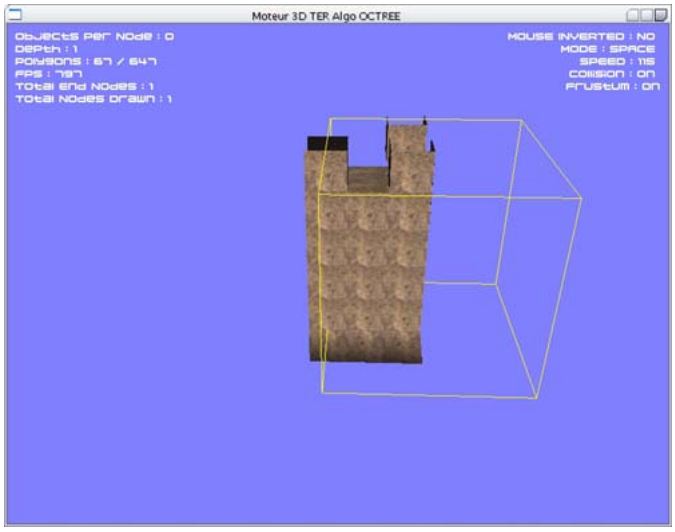
Reprenons l'exemple de la tour (toujours les deux modes de rendu) :



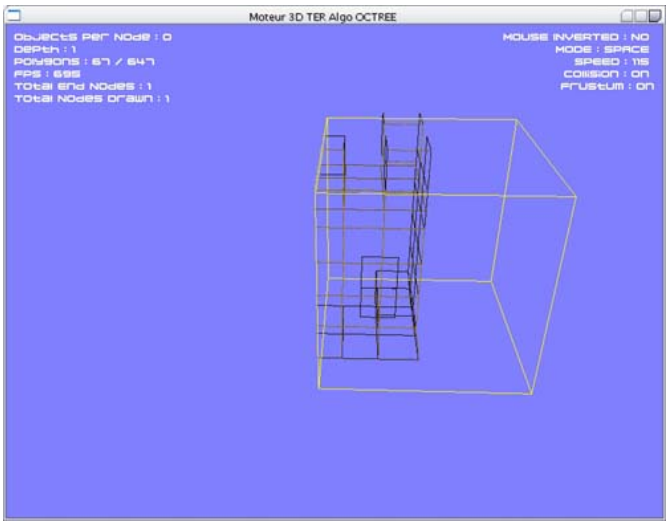



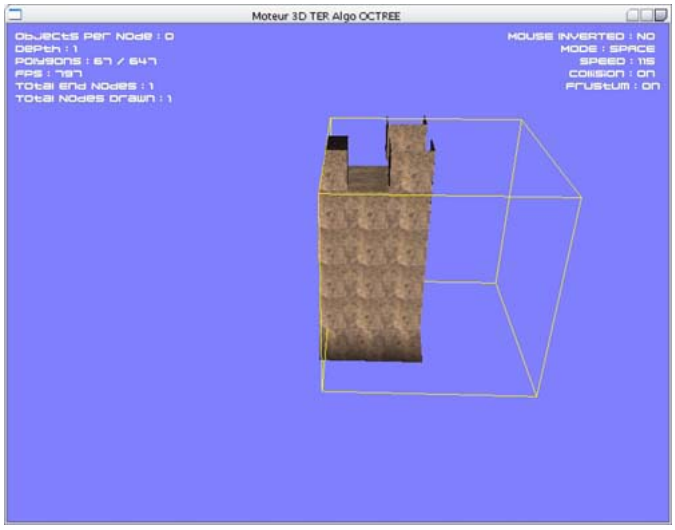

Supposons que l'on garde seulement la partie détourée en vert on obtient ceci :




On observe aisément que des polygones sortent du cube. Notre objectif **aurait était** de couper les polygones pour obtenir ça (retouché avec Photoshop) :



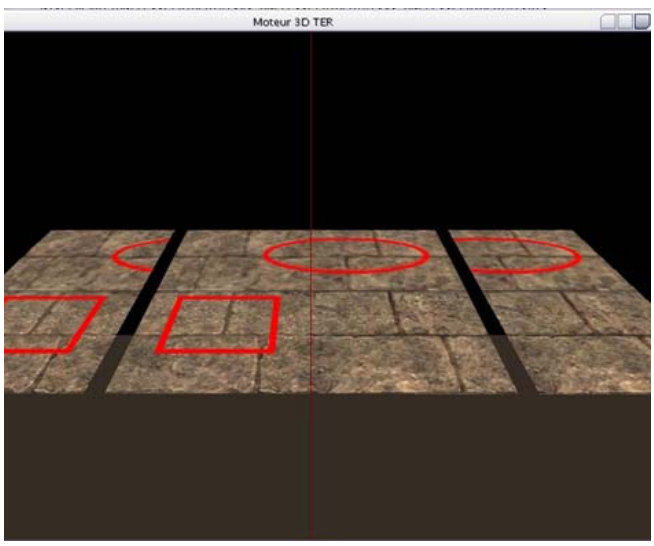



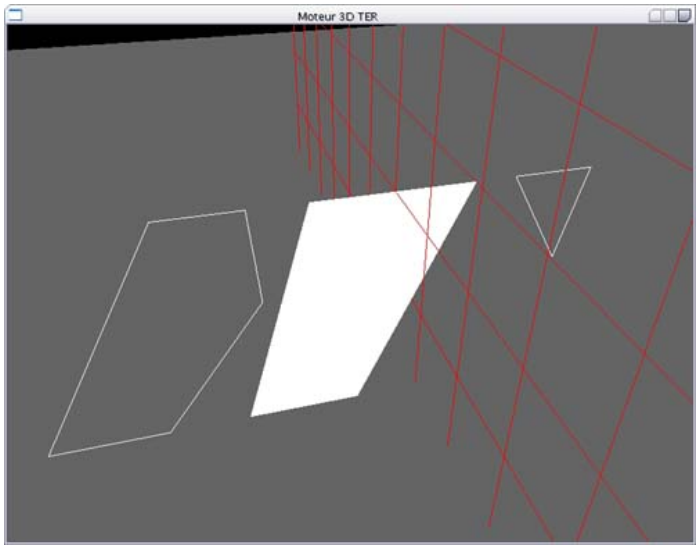
## Tentative de clipping avec l'algorithme de Sutherland-Hodgman

La première étape était de partir d'un plan et de son équation ainsi que d'un polygone avec sa liste de sommets et d'effectuer ce clipping d'après l'algorithme de Sutherland-Hodgman.

Jean-francois Fourmond a implémenté dans un programme à part cet algorithme en mode fil de fer.

j'ai donc récupéré ce programme et j'ai ainsi rajouter le découpage des texture.
En théorie et en pratique ça marche on obtient ça :




On remarque alors dans la figure de droite que lorsqu'une face carré est traversée par le plan dans un ce ses coins, on obtient un polygone à 5 cotés et un triangle (c'est pourquoi il est non texturé). Notre moteur ne stocke que des faces carrés et des faces triangulaires donc en reliant deux sommets de ce polygones on aurait obtenu une face triangulaire et une face carré.

Dans le pire des cas on génère trois faces à partir d'une seule sinon deux, ce qui augmente considérablement le nombre de polygones plus le niveau de profondeur est bas c'est à dire plus le nombre de cubes est important.

## Adaptation avec les octree

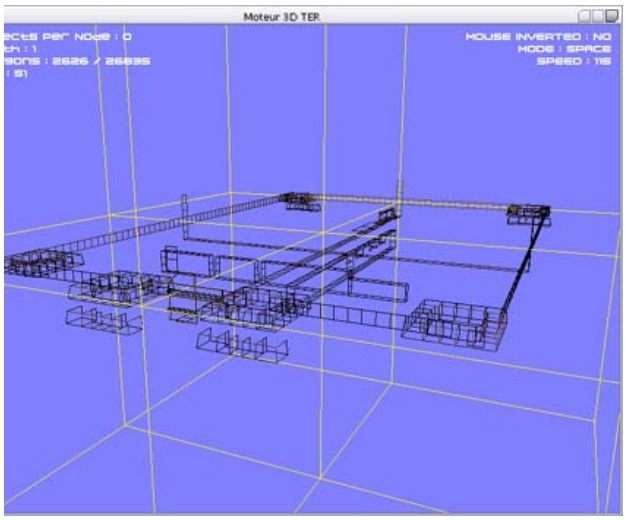
Le clipping jusque la parait fonctionnel mais ne reste pas convainquant quant aux performances qu'il engendrerait; malgré ça il aurait apporté un effet assez plaisant lors de la simulation d'éclatage de notre monde que nous verrons par la suite.
La mise en place du clipping a engendré certaines modifications à notre structure de départ. En effet, nous ne pouvons plus travailler avec des pointeurs sur les objets du fait que ces objets vont être sujet à des modifications spécifiques aux cubes dans

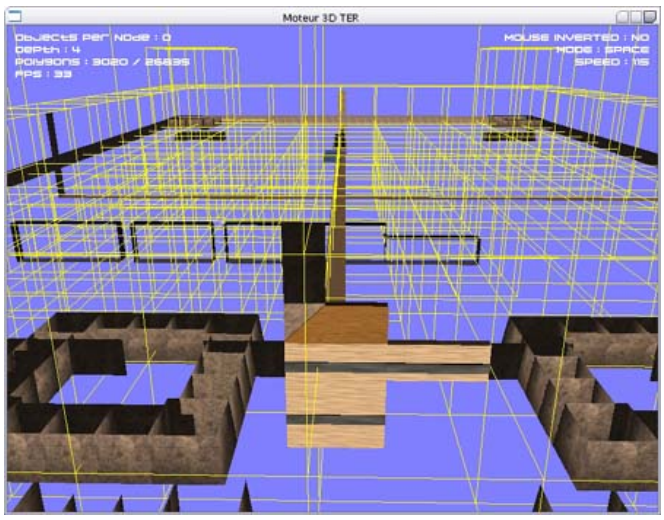
lequel ils sont présents. Nous verrons dans la partie consacrée à l'implémentation des octree quelles modifications structurelles ont été apportées et pourquoi.

La première étape de la mise en place du clipping de polygone dans notre moteur a été d'affecter à chaque nœud terminaux de l'arbre, autrement dit les feuilles contenant les objets, les six équations de plans de ses six faces.
Deuxième étape pour chaque noeud terminal, et pour chaque objet, on parcoure un chaque face de sa liste et on procède à son traitement et ce en fonction des six plans du cube :

- si la faces est à l'extérieur du cube on ne la garde pas.
- si la face est à l'intérieur du cube on la garde sans la couper.
- si la face intersecte un plan, alors il faut la couper.

La détection des faces traversant les plans de cube fonctionne.
Voici deux images de notre monde dans lesquelles seules les faces qui chevauchent au moins deux cubes sont affichées.
Celle de gauche en profondeur 1 et non texturée, celle de droite en profondeur 4 et texturée.




Suite à de longues heures d'implémentations et bien que théoriquement le clipping de polygones était possible, la poursuite de cette implémentation n'a pu arriver à terme; en effet, après notre rendez-vous avec Michel Buffa, il s'est avérer que la mise en place du clipping de polygones n'était pas nécessaire (nous ne l'avions pas cité dans le cahier des charges) et ainsi certains facteurs comme le manque de temps nous a incité à stopper en cours notre progression.
Cependant son implémentation a été réalisée et la majeure partie restante du travail se situe dans la correction d'erreurs.

- ## Classes et Structures

La Classe **Octree** :

Voici comment elle est structurée :

```
Class Octree {
   private:

    // tableau des 8 branches descendantes du noeud courant
    Octree * tabOctreeNodes[8];

    // permet de savoir si on a subidvisé le noeud courant ou pas
    bool subdivided;

    // taille du cube pour le noeud courant
    float widthNode;

   // Centre (X, Y, Z) du noeud courant
   Coord3D centerOfNode;

   // stocke les objets qui devront etre affichés avec ce noeud
   Object3D * m_pObject3D;

   // le dessin du cube représentant l'octree du noeud
   VisualOctree visualOctreeOfNode;

   // tableau des 6 plans du cube du noeud
   Plane tabPlanesOfNodes[8][6];
};
```

En étudiant cette structure on remarque que l'on stocke dans un octree **non pas un pointeur** sur un objet mais un objet ce qui n'est pas du tout optimal, ni en mémoire, ni en temps d'exécution. En effet, cette modification résulte de la tentative de clipping de polygone vue précédemment qui modifie les objets.
Il est donc impossible de travailler avec des pointeurs sur les objets puisque l'on stocke dans les nœuds des objets différents.
Il est important de savoir ou est ce que l'on va stocker les objets.
On peut les stocker soit dans les nœuds terminaux soit dans chaque nœud de l'arbre.
Nous avons choisi dans notre implémentation de stocker les objets dans les feuilles de l'arbre, il n'y a pas de raison précise.

Nous allons à présent expliquer grossièrement comment on créé notre arbre :

- Etape 1 : Créer la racine a partir d'une liste d'objets

Afin de commencer le processus récursif il nous faut un point de départ, la racine de l'arbre.
On créé donc une variable de type Octree sur laquelle on peut lancer la récursivité, en appelant la fonction principale qui créée l'octree à partir de la liste de tous les objets de la scène.

- Etape 2 : Vérification du cas d'arret

Doit on stopper la récursivité ou pas ?
On a choisi comme cas d'arret de fixer à l'avance la profondeur à laquelle on souhaite créé l'octree. La récursivité va continuer tant que ce niveau de profondeur n'aura pas été atteint.

De même on teste le nombre d'objets que contient le nœud, s'il est inférieur au nombre d'objet maximum que l'on a défini à l'avance, on arrête la récursivité (Par exemple si on souhaite au minimum 20 objets par nœuds, on ne pourra pas subdiviser et obtenir des nœuds avec moins de 20 objets).
Si ces deux tests ne sont pas validés, on s'arrête et on stocke la liste des objets avec laquelle on est entré, dans le tableau d'objets du nœud sur lequel on est en train de travailler.

- Etape 3 : Création des listes d'objets pour les 8 nouveaux nœuds

C'est ici que l'on va définir si oui ou non un objet est dans un nœud ou pas.
Pour cela, on aura préalablement créé pour chaque objet sa bounding box (cube englobant) au chargement de notre monde. Pour chaque nœud de l'arbre des octree on connaît les coordonnées des sommets du cube (grâce aux coordonnées du centre et de la taille du cube). Ceci nous permet donc d'être en mesure de savoir si un objet se trouve à l'intérieur d'un cube ou pas.
Dans l'hypothèse où l'objet est dans le cube on ne va pas le stocker entièrement, on parcours la liste des faces de l'objet et on fait le traitement qu'on a vu dans la partie consacrée au clipping :

- si la faces est à l'extérieur du cube on ne la garde pas.
- si la face est à l'intérieur du cube on la garde sans la couper.
- si la face intersecte un plan, on la garde sans la couper.

Lorsqu'une face chevauche deux cubes, elle se trouve donc dupliquée, ce n'est pas très optimal et dans une version ultérieure il sera nécessaire de corriger ce point (par exemple travailler avec des pointeurs sur objet et placer un booléen qui nous dit si une face a déjà été affichée par un autre nœud ou pas).

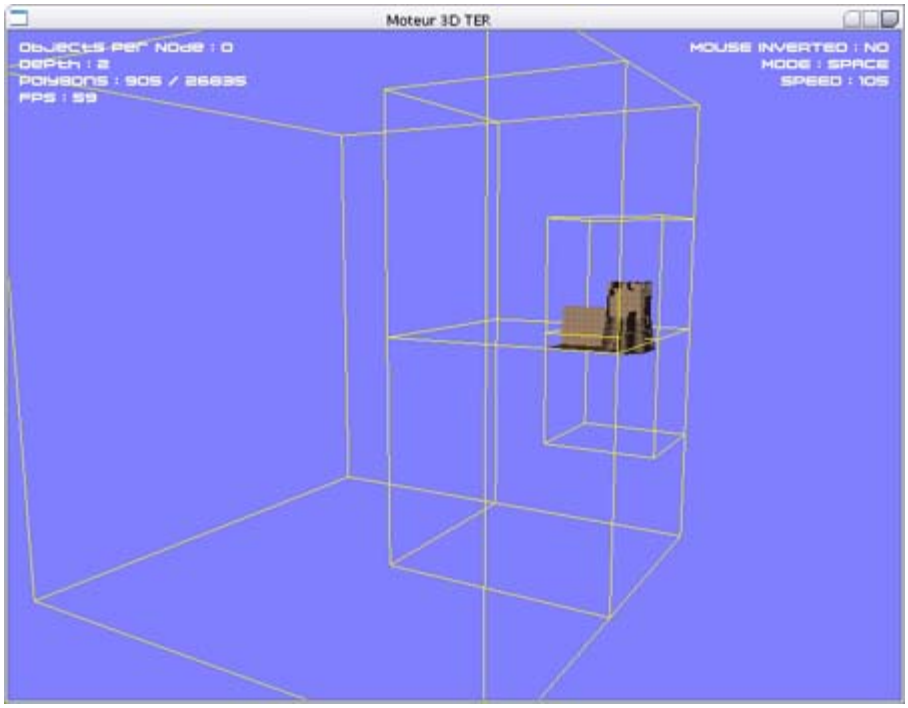
On parcourt donc tous les objets de notre scène et après traitement des faces, on distribue dans chacune des huit listes correspondant au huit nouveaux nœuds, les objets qui apparaissent respectivement dans les huit cubes.

- Etape 4 : Création des nouveaux nœuds / lancement de la récursivité

A partir de la on possède les huit listes correspondant aux huit sous nœuds.
Il nous reste plus qu'a créé ces nœuds en leur attribuant à chacun leur liste respective.

Il est évident que si une liste est vide on ne lance pas de récursivité avec la liste vide, sinon on lance la récursivité pour chacun de ces nœud en retournant à l'étape 1.

Voici une image mettant en valeur la récursivité.




On autorise seulement le stockage dans les cubes "TOP RIGHT FRONT" et "BOTTOM RIGHT FRONT".
Toutes la map a été chargée mais seuls les deux cubes souhaités sont remplis.

La Classe **VisualOctree** :

Cette classe sert uniquement au stockage des lignes définissant les cubes des noeuds.
Voici sa stucture :

```
class VisualOctree {
    private:
```

```
        // liste des lignes
        vector<Coord3D> vLines;
    };
```

On remarque dans la structure Octree, la présence de la donnée membre visualOctreeOfNode

Cette donnée va contenir le dessin du cube (très utile pour du débugage). En effet lorsqu'un nœud terminal (une feuille) est créé, on ajoute les lignes représentant le cube du nœud dans la liste de visualOctreeOfNode.
Ainsi a chaque affichage d'un nœud on a la possibilité ou non d'afficher le cube englobant.

- ## Changement de profondeur

L'intérêt de l'implémentation d'un moteur basé sur l'algorithme des octree est le fait de pouvoir régler le paramètre profondeur afin d'être en mesure d'observer les changements de performance.

Dans l'absolu deux méthodes se sont présentés afin de rendre le paramètre ***profondeur*** réglable :
- Soit on fixe une profondeur de départ et ainsi à chaque utilisation on construit un arbre de même profondeur.

**Avantage :** On créé l'arbre qu'une seule fois donc le changement visuel de profondeur est situé au niveau du parcours de l'arbre. Ainsi le changement de profondeur est instantané.

**Inconvénient :** La profondeur est fixée au départ, on se retrouve coincé entre deux bornes.

- Soit on recréé un nouvel arbre a chaque changement de profondeur.

**Avantage :** On peut créé des arbres de profondeur infinie (en supposant que les performances de la machine nous le permettent)

**Inconvénient :** L'arbre est recréé à chaque fois et pour une scène disposant d'un très grand nombre de polygone et selon le niveau de profondeur il faut attendre que l'arbre se construise.

***Résultats :***

Au regard des deux méthodes présentées ci-dessus, et du fait que Jean-Francois Fourmond a implémenté l'algorithme des quadtree basé sur la première méthode, il me paraissait intéressant d'appliquer la seconde méthode afin de comparer les avantages et inconvénients respectifs.

On rappelle par ailleurs que notre implémentation à but pédagogique nous incite à switcher entre les différents niveaux de profondeur, mais dans un jeu de type FPS, l'arbre est créé une seule fois, et ainsi il n'est pas reconstruit au cours de l'exécution.

### o Coté code

Afin de travailler toujours sur la même variable, nous avons choisi de placer l'initialisation ainsi que la destruction des données des Octree dans les fonctions respectives InitOctree(), DestroyOctree() et non dans le constructeur, destructeur de base de la Classe octree.

De se fait, lors d'un changement de profondeur, il suffit d'appeler les méthodes DestroyOctree() qui libère les zones mémoires allouées au données membres de l'octree racine, en particulier la zone mémoire allouée au stockage des objets de chaque nœud ainsi que la zone mémoire allouée pour tous les nœuds partant de la racine; puis la méthode InitOctree() qui réinitialise certaines données membre.
Il ne suffit plus qu'a relancer la méthode de création des nœuds avec un niveau de profondeur différents.

# Gestion des collisions

## • Introduction

Dans cette partie nous allons voir comment utiliser les octrees pour détecter les collisions.

Il ne faut pas oublier qu'un octree ne sert pas juste au rendu et peut être utilisé aussi bien pour la détection de collision.
Dans notre implémentation, les collisions ne sont pas gérées physiquement au sens propre du terme, c'est à dire que lorsque la caméra se trouve en intersection avec un mur par exemple, elle n'est pas bloquée.

En fait les termes appropriés dans notre cas sont **détection de collision** c'est à dire que en temps réel lorsque l'utilisateur se déplace sur la scène, nous seront en mesure d'observer quelles sont les polygones qui potentiellement peuvent être en collision, et de plus les faces qui sont réellement en collision avec notre caméra.

La gestion des collisions fait appel à des notions mathématiques parfois un peu complexes, et ainsi les différentes fonctions de base comme les fonctions **d'intersection plan-sphère**, ou **intersection sphère-polygone**, ou autre fonctions de géométrie spatiale n'ont pas été implémentées de zéro mais ont été trouvés sur divers site consacré à la programmation des jeux.
Ces fonctions ont été largement commentées dans le code, et de plus nous allons les étudier dans ce qui suit.
Le travail a donc consisté à les implanter avec les octrees ce qui n'est pas une mince affaire.

- # Première étape - la caméra

Pour la suite nous allons devoir schématiser le corps que l'on veut déplacer dans notre scène par une sphère, en l'occurrence notre caméra.
La sphère reste une assez bonne approximation (en terme de volume et de forme) pour la plupart des objets (joueur ou voiture par exemple).
On englobe notre caméra par cette sphère en définissant une donnée membre **radius** dans notre Classe Caméra, qui sera le rayon de la sphère.
Nous avons choisi de placer ce paramètre réglable par l'utilisateur afin de d'être en mesure de visualiser les différences de comportement selon la taille de la sphère.

- # Octree & détection des collisions

Les travaux vont s'orienter principalement autour de deux éléments **l'arbre des octrees** et **la caméra**.
L'objectif de la manipulation est de faire appel, à chaque rendu, à une fonction qui va utiliser l'arbre des octrees en fonction de la camera. Pour se faire on va créé une fonction *OverlappsCamera(Camera *cam )* que l'on appelle sur la racine de l'arbre.

***Principe* : Si la *sphère englobante de la caméra* est présente dans un octree, il y a possibilité quelle soit en collision avec les polygones de cet octree, et il est impossible qu'elle soit en collision avec les polygones des autres octrees. Les test de collisions seront donc fait seulement sur un nombre de polygones précis (ceux des octrees concernés) et non sur tous les polygones de la scène. De même que pour l'affichage, le nombre de polygones à tester est considérablement restreint comparé au nombre de polygones total composant notre monde.**
**C'est dans ce principe que réside l'intérêt du partionnement d'espace.**

  - **Etape 1 - Vérifier la position de la camera**

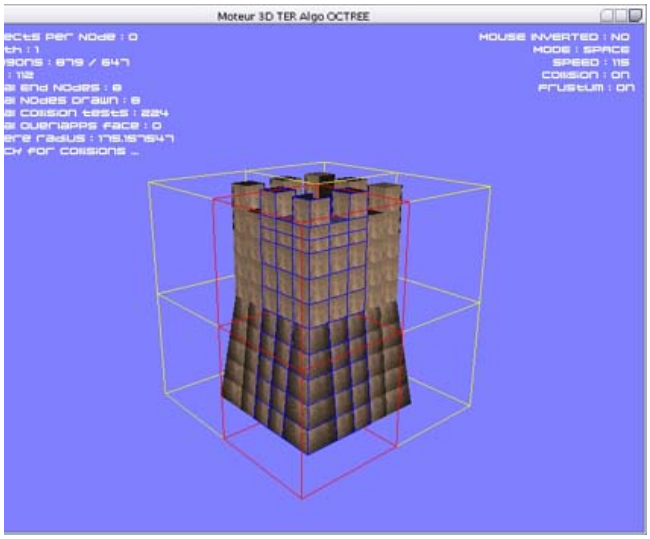
Au départ, on lance la fonction sur la racine de l'arbre. On rappelle que la racine de l'arbre est un nœud auquel on associe un cube englobant toutes la scène.
On va donc vérifier si notre caméra est dans ce cube ou pas. Il faut bien avoir à l'idée que si la caméra n'est pas présente dans ce cube, il n'y a absolument **aucune chance** qu'elle soit en collision avec les polygones de ce cube. Pour cela on va utiliser la fonction *IsOverlappsCameraWithCube()* qui prends en paramètre les coordonnées du centre du cube octree sur lequel on fait les tests d'intersection avec la sphère ainsi que la taille du cube.

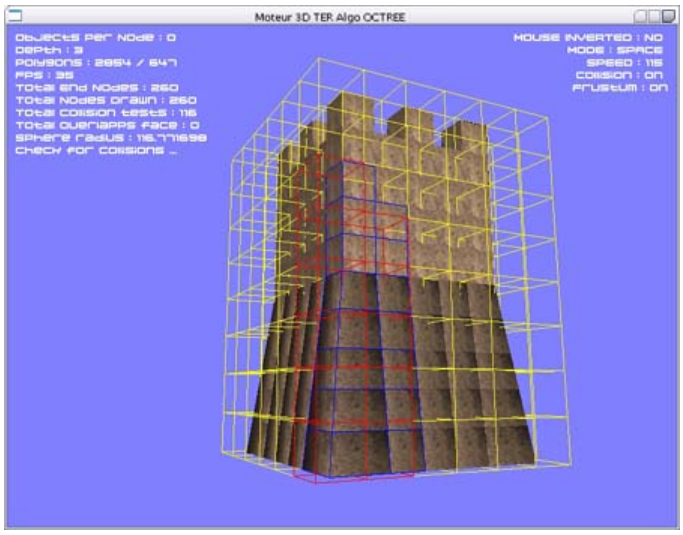
Si un cube intersecte avec notre caméra alors les polygones de ce cube deviennent potentiellement aptes à être en collision avec notre sphère => on va alors à l'étape 2.
Sinon si le nœud est subdivisé alors on lance la récursivité sur chacun de ses fils puisque les polygones sont stockés dans les feuilles.

  - **Etape 2 - Détecter les polygones**

Si l'on est ici c'est que la sphère englobante de la caméra intersecte avec un cube.

***Tous*** les polygones sont donc potentiellement suspect pour la collision. Pour obtenir une vue éloignée afin de bien comprendre, le rayon de la sphère a été très largement augmenté.
Voici deux images de profondeur différentes (la première de profondeur un, l'autre deux) afin d'illustrer ceci.
En rouge les cubes concernés, c'est à dire ceux qui intersectent avec la sphère englobante de la caméra.
En bleu les polygones suspects.
[la couleur des polygones est plus significatif en mode texturé]




**REMARQUE :** Les images n'ont pas été retouchées avec un logiciel de dessin, les polygones se détourent de la couleur appropriée en temps réel.

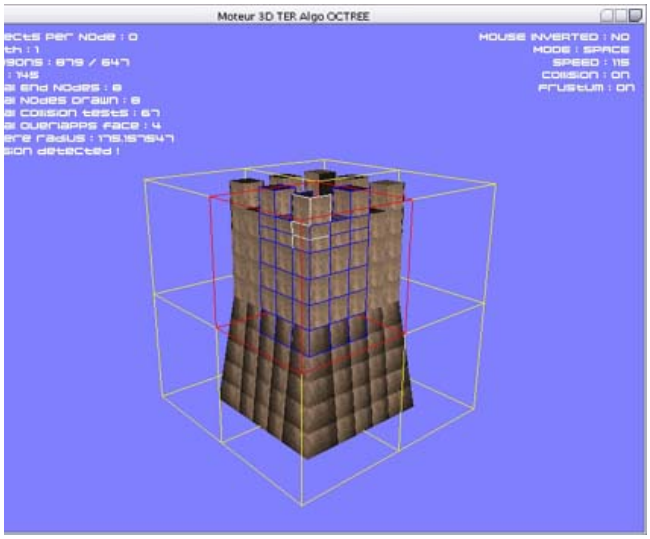
[image de gauche]
- 224 polygones testés
- 0 en collision avec la sphère
- rayon de sphère : 175,15

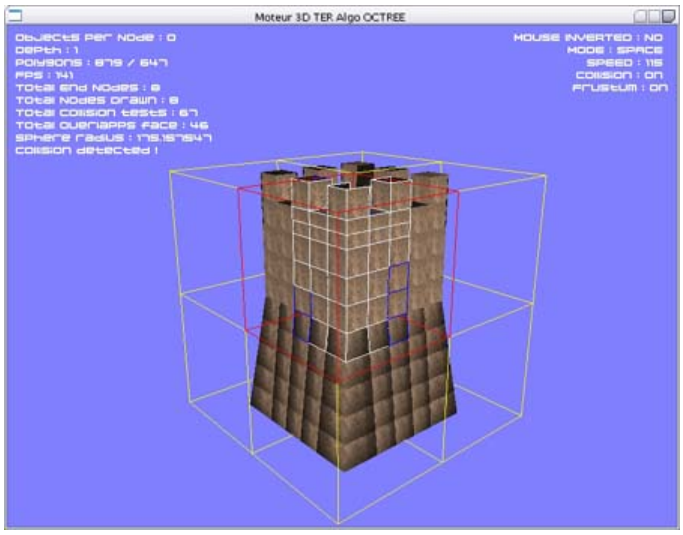
[image de droite]
- 260 polygones testés
- 0 en collision avec la sphère
- rayon de sphère : 175,15

On observe bien que les polygones bleus correspondent à tous les polygones du cube en rouge.

### o Etape 3 - Tester les collisions sur les polygones

Maintenant que l'on connaît les polygones qui peuvent être en collision avec la sphère, il ne "suffit" plus que de tester si il y a effectivement collision ou pas. Pour cela on a la fonction *IsOverlappsCameraWithFace()* qui renvoie vrai si la sphère est en collision avec le polygone passé en paramètre, faux sinon.
Dans l'hypothèse ou il y a intersection on dessine ces polygones en blanc.
Dans les images qui suivent l'image de droite est la même sauf que l'on a légèrement avancer.

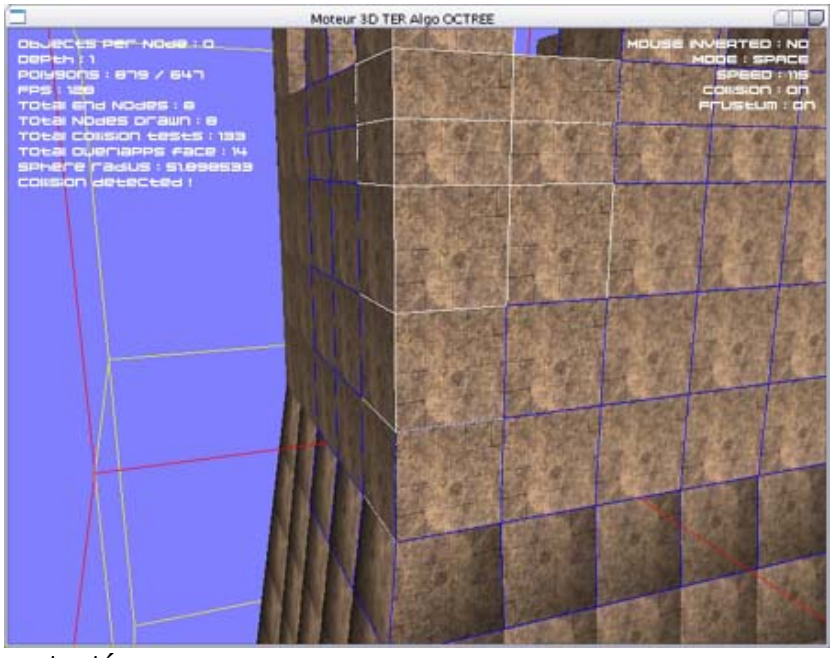



[image de gauche]
- 67 polygones testés
- 4 en collision avec la sphère
- rayon de sphère : 175,15

[image de droite]
- 67 polygones testés
- 46 en collision avec la sphère
- rayon de sphère : 175,15

On observe que le nombre de polygones blancs (en collision) a augmenté. Dans celle-ci on a réduit la taille de la sphère et on s'est rapproché pour bien observer la couleur dont chacun est détouré.

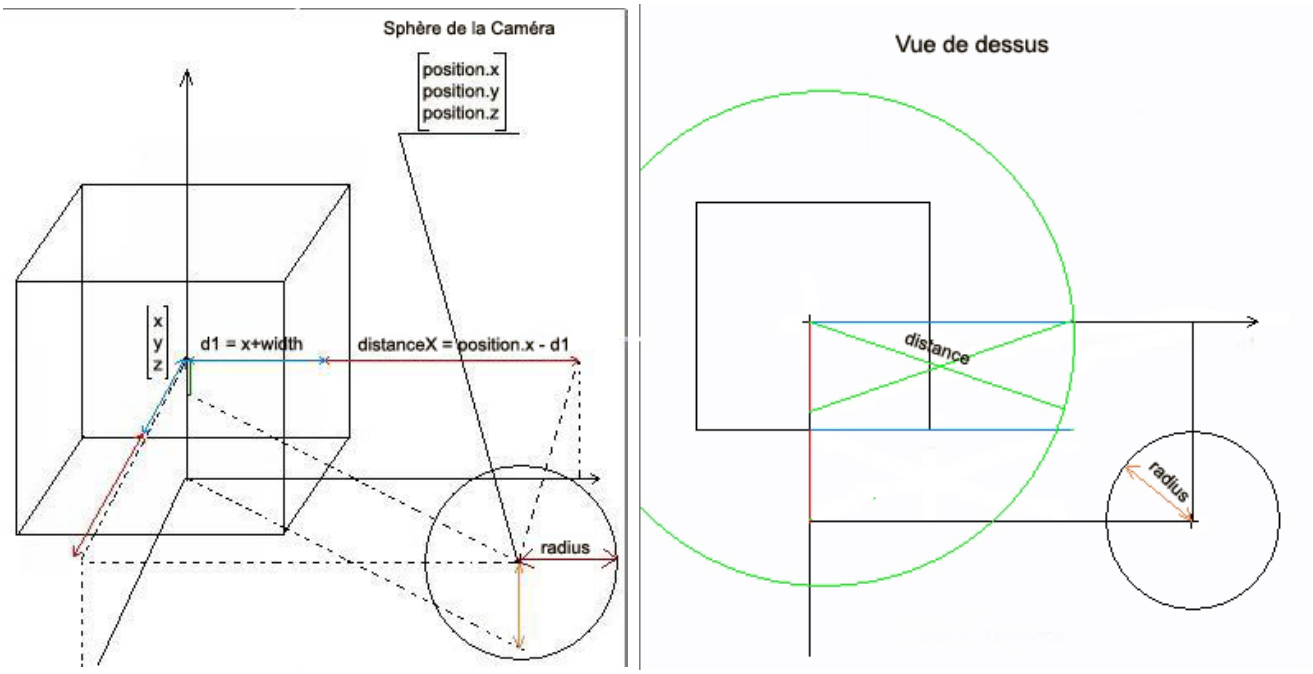



- 133 polygones testés
- 14 en collision avec la sphère
- rayon de sphère : 51,89

- # Fonctions utilisées

    - **bool IsOverlappsCameraWithCube()**

Comme nous l'avons vu cette fonction permet de savoir si il y a intersection entre un octree (disons son cube) et la sphère de la caméra. Nous avons en paramètres les coordonnées du centre du cube (x, y, z) ainsi que sa taille (width = taille / 2).
[Paint inside]




[Vue 3D]
[en pseudo code]
si (position.x > d1) alors
    distanceX = position.x - d1
si (position.y > d2) alors
    distanceY = position.x - d2
si (position.z > d3) alors
   distanceZ = position.x - d3
si (distanceX²+distanceY²+distanceZ²) < radius² alors il y a intersection entre le cube et la sphère.

Le pseudo code écrit ci-dessus s'applique dans le cas de figure du schéma.
Il est évident que dans l'hypothèse ou la sphère est de l'autre coté du cube il est nécessaire d'inverser les soustractions.

Voici l'explication en 2D qui sera peut être plus explicite.

[Vue 2D]
Soit distanceX le segment bleu décalé au centre.
Soit distanceY le segment rouge décalé au centre.
alors distance² = distanceX² + distanceY² (pythagore).
On voit nettement que si (distance < radius) alors il y a intersection entre le carré le cercle.
En fait le fait de calculer (distance² < radius²) nous permet d'éviter de calculer la racine carré de distance.
En fait il est plus optimal d'élever *radius* au carré puis de faire la comparaison plutôt que de chercher la racine carré de *distance*.
On renvoie donc vrai si (distance² < radius²) faux sinon.

- **bool IsOverlappsCameraWithFace()**

Cette fonction présente dans la classe caméra délègue le calcul à la fonction bool *SpherePolygonCollision()* , c'est celle ci que nous allons étudier.

- **bool SpherePolygonCollision()**

Cette fonction retourne vrai si la sphère intersecte le triangle passé en paramètre.
Il s'agit de la seule fonction à appeler pour vérifier si une sphère est en collision avec un polygone.
Voici les différentes étapes de recherche :

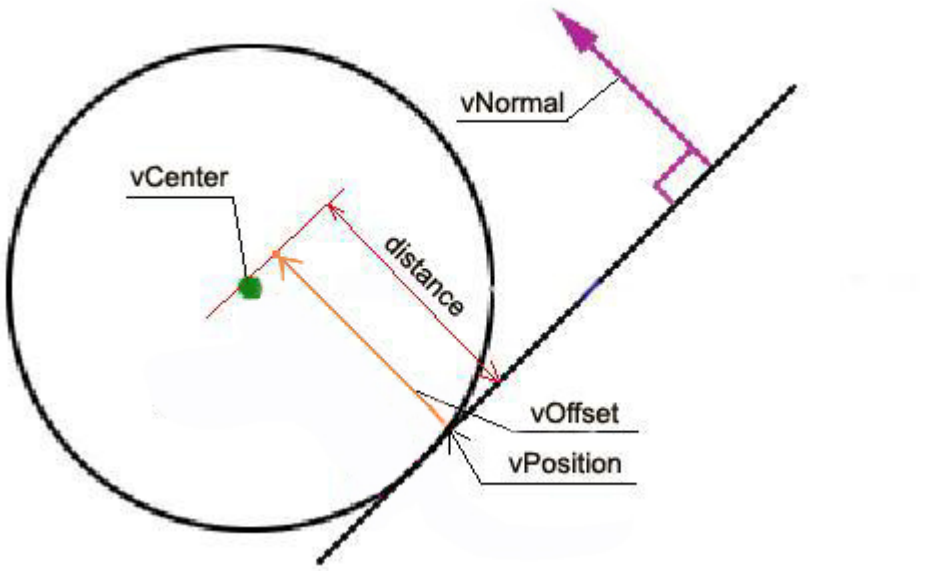
- **Etape 1 - Positionner la sphère par rapport au plan du polygone**

Tout d'abord rappelons que les plans sont infinis.
Il nous faut une fonction qui renvoie la position de la sphère par rapport au plan afin de savoir si elle est complètement devant, complètement dérriere ou en intersection avec ce plan.
On a donc la fonction int *ClassifySphere()* qui retourne BEHIND, FRONT ou INTERSECTS.
Si *ClassifySphere()* retourne INTERSECTS, alors on va à l'étape 2, sinon la sphère n'est pas en collision avec le polygone.

- **Etape 2 - Trouver le point d'intersection entre la sphère et le plan du polygone**

Si on est ici, on peut affirmer qu'il y a intersection entre la sphère et non pas le polygone mais le plan du polygone.
L'objectif est donc de savoir si le point d'intersection est à l'intérieur du polygone ou à l'extérieur. Pour cela on va utiliser la fonction bool *InsidePolygon()* qui prend le point d'intersection et le triangle et qui renvoie vrai si le point est à l'intérieur du périmètre du polygone, faux sinon.
Mais avant tout il faut récupérer ce point d'intersection.
Comme la sphère possède un nombre infini de points, il peut y avoir un million de points en collision avec.

On sait obtenir le vecteur normal d'un polygone ce qui va nous permettre de savoir de quel coté est tourné le polygone.
On a la fonction qui renvoie la distance du centre de la sphère au plan ( *ClassifySphere()* met à jour la variable distance passée en référence) et la normal nous donne l'orientation du plan.
Voici un schéma pour mieux comprendre : [Paint inside]

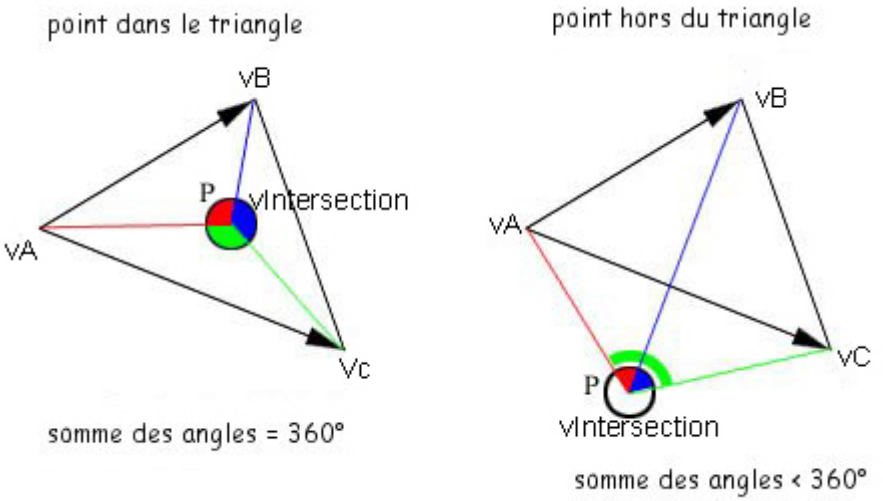



En fait on projette le centre de la sphère sur le plan dans la direction de la normale.
On le fait en multipliant la normale par la distance du centre de la sphère au plan.
On obtient ainsi un vecteur qu'on nommera vOffset sur le schéma d'où :
vOffset = vNormal * distance
ensuite on veut récupérer le points d'intersection sur le plan d'ou
vPosition = vCenter - vOffset

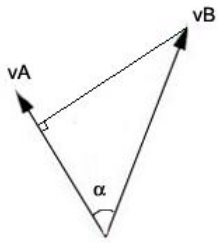
- **Etape 3 - Le point d'intersection est-il dans le périmètre du polygone ?**

Une fois qu'on a le point d'intersection, on doit regarder si le point d'intersection est à l'intérieur du polygone ou pas.
Si c'est le cas, c'est qu'il y a collision entre la sphère et le polygone sinon on passe à l'étape 4 pour une dernière tentative. Comment savoir si le point d'intersection trouvé appartient au polygone ?
C'est le rôle de la fonction bool *InsidePolygone()*.
On sait que le point appartient au même plan que le polygone donc on peut réduire le problème en 2D.

Voici un schéma qui rend le problème trivial :




Comme on peut le constater il suffit de faire la somme des trois angles et regarder si elle est égale à 360 degrés ou 2PI.
Pour cela on créé les vecteurs vA-vIntersection, vB-vIntersection et vC-vIntersection puis on fait appelle a une fonction qui calcul l'angle entre les vecteurs deux à deux.
Le calcul de l'angle entre deux vecteurs se retrouve facilement :
[voici deux vecteurs, on cherche alpha – [Paint inside)]



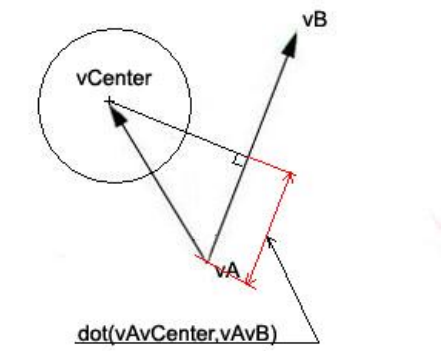

(norme(vA)*norme(vB)) * cos(alpha) = produitScalaire(vA,vB)
d'ou
alpha = acos(produitScalaire(vA,vB) / (norme(vA)*norme(vB)))
Donc dans l'hypothèse où le point d'intersection est dans le périmètre du polygone il y a collision. Sinon on passe à l'étape 4.

- **Etape 4 - Vérifier si la sphère intersecte avec les arêtes du polygone**

Ici le point d'intersection n'est pas dans le périmètre du polygone.
Comme on travaille avec une sphère, on peut encore être en collision à cause du rayon de la sphère.

Pour cela il faut calculer le point le plus proche appartenant à une arrête du polygone. Ceci est fait en prenant le point pour lequel la distance 'point d'intersection -- point le plus proche sur une arrête du polygone' est minimale. Il va falloir faire ceci sur chacune des trois arêtes du triangle.

La fonction bool *EdgeSphereCollision()* renvoie vrai si un point de la sphère intersecte avec une des arêtes du triangle, faux sinon.
Si n'importe quelle partie de la sphère intersecte avec les arêtes du polygone alors il y a collision.
Ceci est fait uniquement si le centre de la sphère est à l'extérieur des arêtes du polygones.
Pour chaque arête du polygone, on doit trouver le point le plus prés du centre de la sphère. Si la distance de ce point est inférieure au rayon de la sphère, il y a collision.
Comment trouver le point d'une arête le plus proche de la sphère ?
[voici un schéma explicatif - (Paint inside)]




Le principe est assez simple : - on créé les vecteurs vA-vCenter et vA-vB.
- et on calcule le produit scalaire des deux vecteurs, ce qui revient à projeter vA-vCenter sur vA-vB.

Résultat :
On voit nettement que si le produit scalaire des deux vecteurs est inférieur à la norme du vecteur vAvB alors le point vA est plus proche de la sphère que ne l'est vB sinon c'est l'inverse.

# Simulation d'éclatage de la scène

Afin de créer cette simulation d'éclatage, on a du faire intervenir dans la structure de l'octree, une nouvelle donnée membre.
on rappelle la structure d'un octree :

```
Class Octree {
   private:

    // tableau des 8 branches descendantes du noeud courant
    Octree * tabOctreeNodes[8];

    // permet de savoir si on a subidvisé le noeud courant ou pas
    bool subdivided;

    // taille du cube pour le noeud courant
    float widthNode;

   // Centre (X, Y, Z) du noeud courant
   Coord3D centerOfNode;

   // decalage du noeud pour l'eclatage de la map
   Coord3D shiftOfNode;

   // stocke les objets qui devront etre affichés avec ce noeud
   Object3D * m_pObject3D;

   // le dessin du cube représentant l'octree du noeud
   VisualOctree visualOctreeOfNode;

   // tableau des 6 plans du cube du noeud
   Plane tabPlanesOfNodes[8][6];
};
```

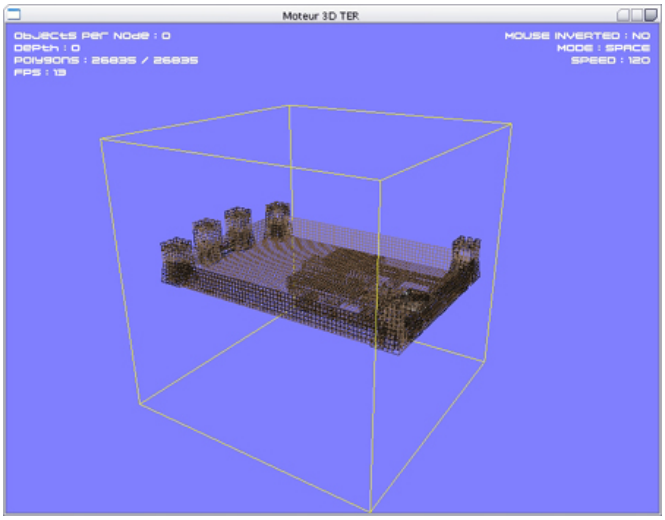
En comparant à la structure présenté précédemment on remarquera la présence de la donnée membre shiftOfNode.
Au même titre que chaque nœud (cube) possède un centre et une taille, il contient de même un vecteur décalage.

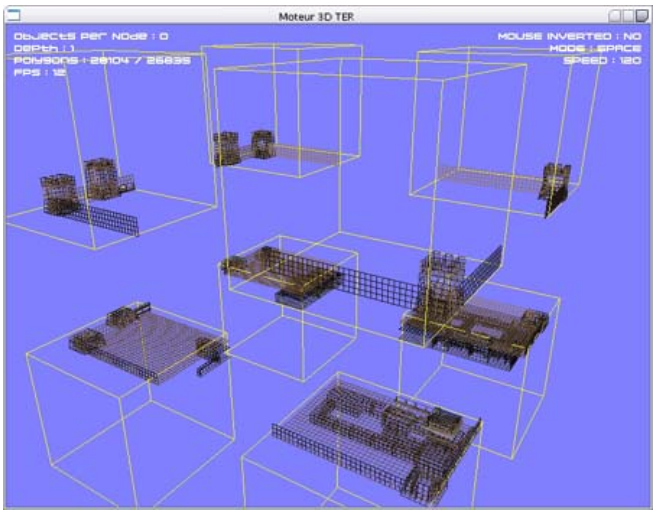
Ce vecteur doit être défini pour chaque octree dans la récursivité lors de la création du nœud du fait qu'il doit tenir compte du décalage de son nœud parent.
On a fixé un décalage de nœud de "width/4" (c'est complètement arbitraire) ainsi plus on plonge dans l'arbre plus l'espace entre les cubes est petit.

A chaque affichage il suffit d'ajouter les coordonnées de cette variable à tous les sommets, qu'il s'agisse des sommets d'un objet ou des sommets des cubes des octrees.

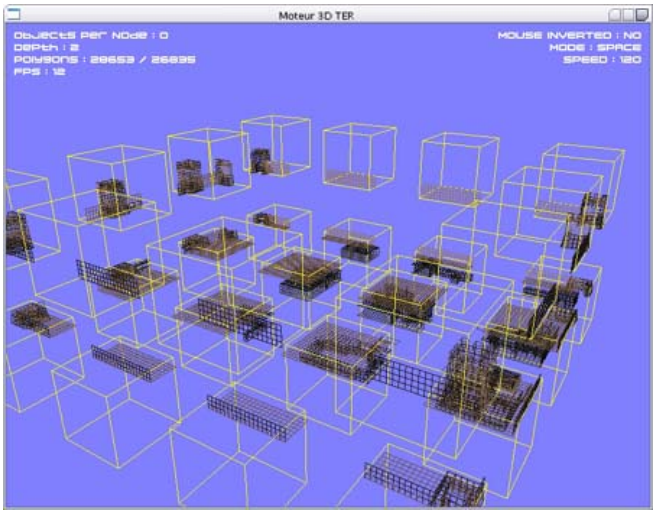
Voici quelques images :

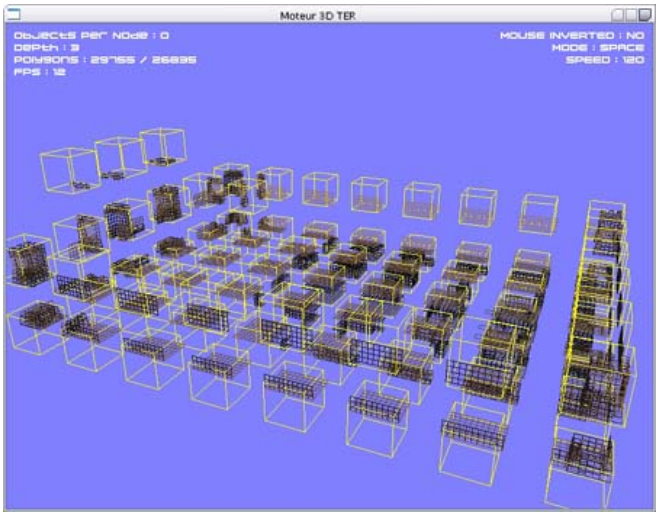
[profondeur 0 et 1]



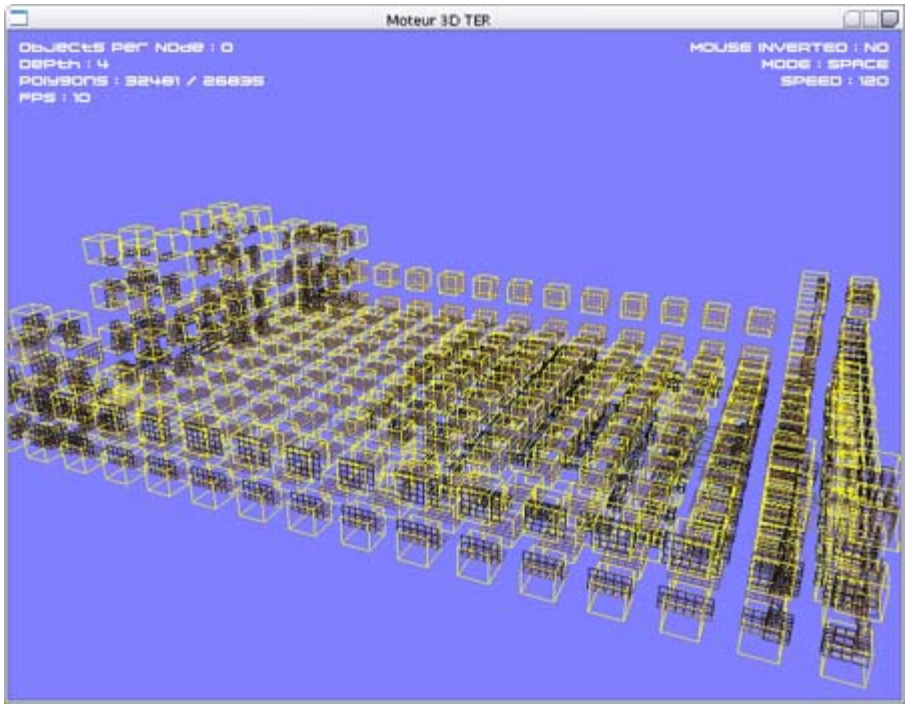




[profondeur 2 et 3]

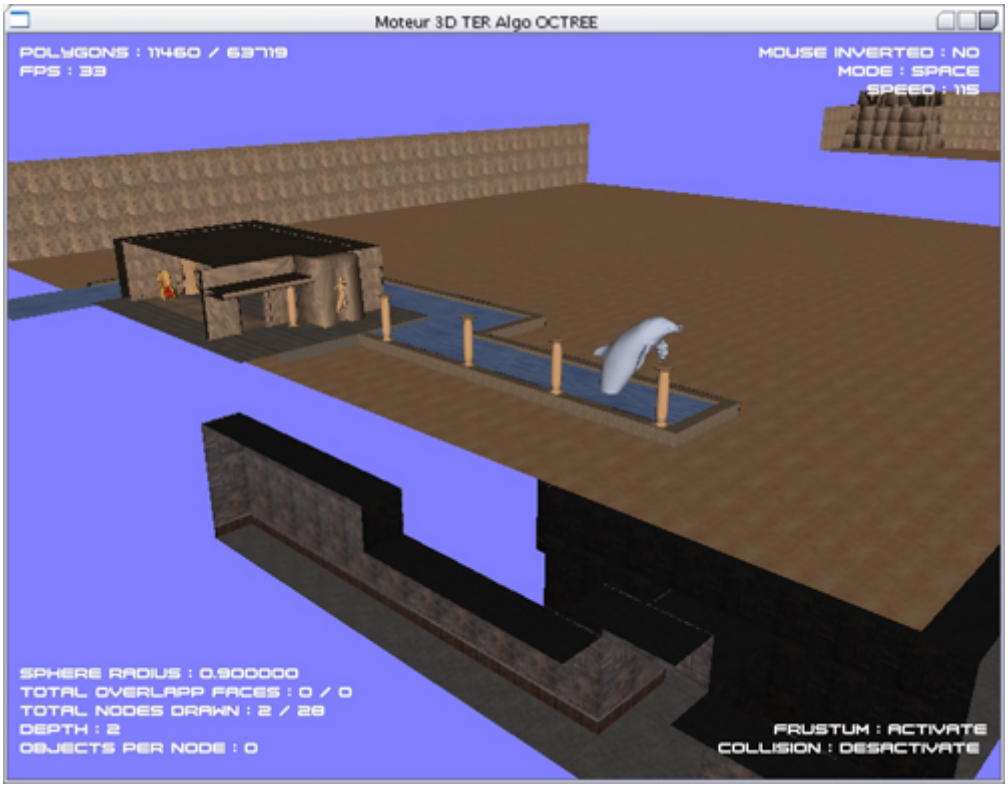


[profondeur 4]




**[ mode texturé réalisé avec la version quasi finale du moteur ]**


Moteur 3D TER Algo OCTREE
POLYGONS : 11460 / 63719
FPS : 33
MOUSE INVERTED : NO
MODE : SPACE
SPEED : 115
SPHERE RADIUS : 0.900000
TOTAL OVERLAPP FACES : 0 / 0
TOTAL NODES DRAWN : 2 / 29
DEPTH : 2
OBJECTS PER NODE : 0
FRUSTUM : ACTIVATE
COLLISION : DESACTIVATE

# Conclusion

- Résultats
    - Coté implémentation :

L'algorithme des octrees a été assez facile à implémenté, sans doute le plus difficile reste son traitement, en particulier la gestion des collisions "physiques" (avec glissement de la caméra lors du blocage sur un mur).
Comme nous l'avons vu, l'épopée du clipping a contraint à un changement de structure, ce qui n'a pas rendu le code optimal lors de la construction de l'arbre, le manque de temps ne nous permet pas de faire les modifications d'ici la remise des livrables, cependant nous avons pris note des changements à apporter pour une version ultérieure.

De même, concernant la reconstruction de l'arbre à chaque changement de profondeur, la méthode utilisé nous permet de ne pas être borné et ainsi de créé des arbres de profondeur illimitée. Nous avons choisi cette méthode afin de différer de celle de Jean-francois Fourmond dans l'étude des quadtrees. Cependant il s'avère que dépassé un certain niveau, le nombre d'octrees étant tellement grand que l'on se retrouve borné par les performances de la machine. c'est pourquoi nous auront beaucoup de mal a dépassé la profondeur 9.

De ce fait la modification de ce point serait à apporter dans une version ultérieure.

- Avantages

L'avantage des octrees dans l'absolu est que l'arbre est pré-calculé, (si nous ne changeons pas de niveau de profondeur dans notre cas) et ainsi les performances de la machine sont concentrées sur son utilisation.
De même la structure hiérarchique des octrees engendrent un découpage du monde rendant le repérage facile dans la scène.

En effet nous avons vu lors de la gestion des collisions qu'avec seulement quelques tests il nous était possible de retrouver la position de la caméra dans la hiérarchie globale des octrees, ceci est très intéressant, en particulier pour la gestion des collisions, dans laquelle on a sans cesse besoin de connaître la position de la caméra par rapport aux polygones de la scène.

- Inconvénients

Il s'avère cependant que les octrees restent non optimaux dans les espaces fermés. En effet nous supposons que notre plan far (le plan du frustum le plus éloigné) est à l'infini, si la caméra se trouve dans un espace clos et un mur est présent devant le champs de vision de la caméra, le découpage de l'espace fait que tous les cubes créés se trouvant derrière ce mur seront affichés en plus de ce mur.
De même comme nous l'avons vu l'algorithme génère de nouvelles faces, et ainsi plus la subdivision est grande plus le nombre de cube est important et plus le nombre de faces augmente. Ceci est assez contraignant lorsqu'on décide de s'éloigner au maximum de la scène avec le même nombre de subdivision, car tous les cubes sont

dans le champs de vision de la caméra et ainsi beaucoup plus de polygones sont affichés comparé au nombre total dont se compose la scène au départ. Ceci nous amène à la partie suivante.

- Extension

En effet, en supposant que l'on découpe la scène et que l'on s'éloigne afin d'avoir toute la scène dans le champs de vision.

Du fait que les polygones chevauchant les cubes sont dupliqués, beaucoup plus de polygones sont affichés dans ce cas que ce que sont affichés sans subdivision. Ainsi intervient la notion de niveau de détail ou Level Of Details (LOD) en anglais. Le but est de stocker dans l'arbre pour chaque nœud des listes correspondantes à des niveaux de détail différents. Ainsi lors de l'affichage on teste l'éloignement de la caméra par rapport à un cube et on affiche la liste de polygones du niveau de détail correspondant. Il est évident qu'une liste de niveau de détail bas contiendra beaucoup moins de polygones qu'une liste de niveau de détail supérieur.
Ceci permet de palier au surplus d'affichage de polygones lorsqu'on visualise la scène entière.

# Les arbres BSP

## Historique

Henry Fuchs, Zvi Kedem et Bruce Naylor ont été les premiers à implémenter cet algorithme. Leur premier article, "On Visible Surface Generation by A Priori Tree Structures" est paru en 1980. A ce moment là, ils utilisaient les arbres BSP pour déterminer l'ordre d'affichage des polygones en fonction de la profondeur pour pallier les faiblesses de l'algorithme du peintre. En effet, il n'y avait pas de carte graphique sur les ordinateurs du début des années 80 et le calcul du depth buffer (tampon de profondeur) faisait chuter les performance.
On a commencé à utiliser les arbres BSP au début des années 90. Le premier jeu à les utiliser et à en faire une implémentation qui donne un rendu fluide est Doom (en 2D), développé par John Carmack et John Romero. Doom étant devenu très populaire dans le monde du jeu vidéo, de nombreux autres jeux se sont mis à utiliser les BSP. Aujourd'hui, de nombreux moteurs de jeux utilisent les BSP : la sage des Quake (Id Software), Unreal (Epic Game), …

## Définition

Un arbres BSP (Binary Space Partition Tree) est une structure de données où sont subdivisés des espaces afin d'isoler ce que l'on pourra appeler des éléments de base. Chacun des nœuds de l'arbre subdivise (de manière hiérarchique) l'espace en deux selon un plan. Le nœud racine subdivise le monde (l'espace) en deux sous espaces, puis chacun des deux fils subdivise à son tour l'un des deux sous espaces en deux nouvelles parties. La division se répète de manière récursive jusqu'à ce que l'on ait attribué un sous espace à chaque élément de base, l'élément de base étant un ensemble de polygones coplanaires (qui sont les faces du monde à afficher). Cette structure de données permet de déterminer les éléments de base qui ne se trouvent pas dans le champ de vision (donc qu'il n'est pas nécessaire d'afficher) en se servant des propriétés de parcours des arbres : possibilité d'éliminer le traitement (coûteux en temps processeur et en ressources graphiques) d'un branche entière de l'arbre sur le résultat d'un test (peu coûteux en temps processeur) sur un nœud de l'arbre.

## Limitation des arbres BSP

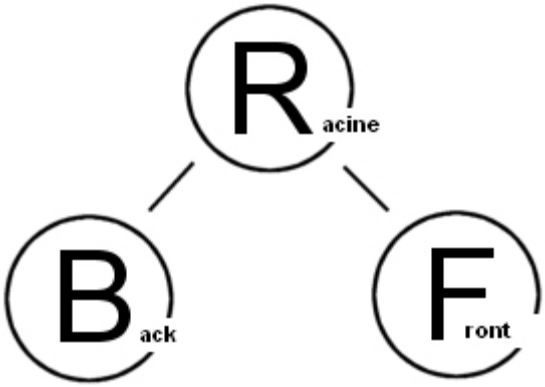
Avant de décrire plus en détail leur fonctionnement, il faut préciser que la réalisation d'un arbre BSP nécessite un temps de calcul si important qu'il n'est pas possible de le construire dynamiquement pendant l'exécution, même en essayant de le calculer au chargement du programme. En effet, il faut plusieurs minutes (on peut même arriver à un temps de calcul de l'ordre de heure, en fonction de la puissance de l'ordinateur et du nombre de polygones constituant le monde) pour construire un arbre BSP de manière efficace. Cette étape est appelée compilation de l'arbre BSP.

# Construction de l’arbre BSP à partir d’un fichier monde (Compilation)

Chaque nœud de l’arbre contient les informations suivantes :

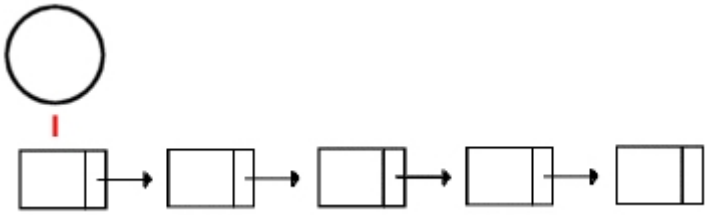
- un plan orienté qui coupe une zone de l’espace en deux (une partie avant et une partie arrière)
- un ensemble de faces portées par ce plan
- une boite englobante (bounding box) que l'on remplira après les calculs des plans de coupe
- un fils gauche et un fils droit de type nœud

La racine de l’arbre représente donc la première subdivision de l’espace.




Par convention, on considèrera que le fils gauche « contient » les faces situées devant le plan défini par le nœud courant et que le fils droit « contient » les faces situées devant le plan défini par le nœud courant.

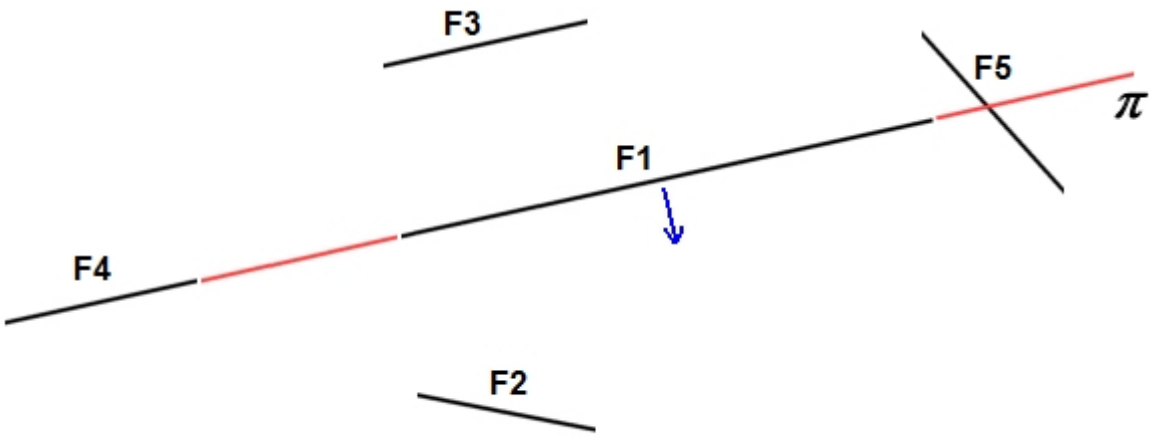
Après le chargement d’un fichier représentant un monde, on se retrouva dans la situation suivante, en supposant que l’on choisisse une liste chaînée dont chaque maillon représente une face :



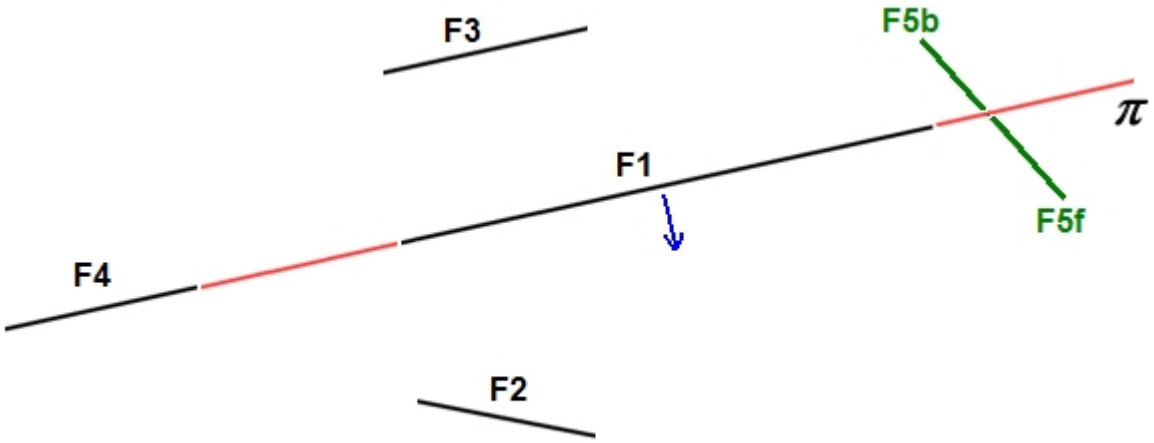
A partir de là, on subdivise le monde en deux selon un plan de partitionnement afin de construire les fils du nœud. Le plan de partitionnement est un plan porteur d’une face du monde. Nous verrons par la suite comment choisir judicieusement un plan de partitionnement.
Une fois ce plan choisit, on distingue quatre types de face :

- les faces coplanaires à ce plan, que l'on stockera dans une liste de faces que l'on appellera «liste associée au nœud»,
- les faces placées derrière le plan de partitionnement sont stockées dans la liste associée au fils gauche,
- les faces placées devant le plan de partitionnement sont stockées dans la liste associée au fils droit,
- et les faces coupées par le plan de partitionnement qui ont à la fois une partie devant et une partie derrière le plan.

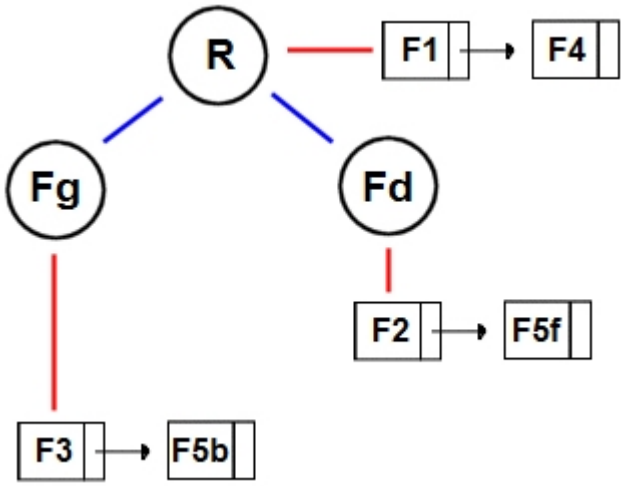
Prenons l'exemple suivant vue de dessus :




Ici, le plan Pi porteur de la face F1 a été choisi comme plan orienté de partitionnement, la flèche bleue indiquant la partie avant du plan. Les faces F1 et F4 sont coplanaires à Pi et sont donc ajoutées à la liste associée à la racine de l'arbre. La face F2 est située devant le plan Pi et est ajoutée à la liste associée au fils droit. La face F3 est située derrière le plan Pi et est ajoutée à la liste associée au fils gauche. La face F5 quand à elle est coupée par le plan Pi dans ce cas, on coupe la face F5 en deux de la façon suivante :



On calcule l’intersection de Pi et de F5 et on découpe F5 en F5f qui est ajoutée à l’ensemble de faces situées devant Pi (liste associée au fils droit) et en F5b qui est ajoutée à l’ensemble des faces situées derrière Pi (liste associée au fils gauche). Après la première subdivision, on a un arbre dans la configuration suivante :

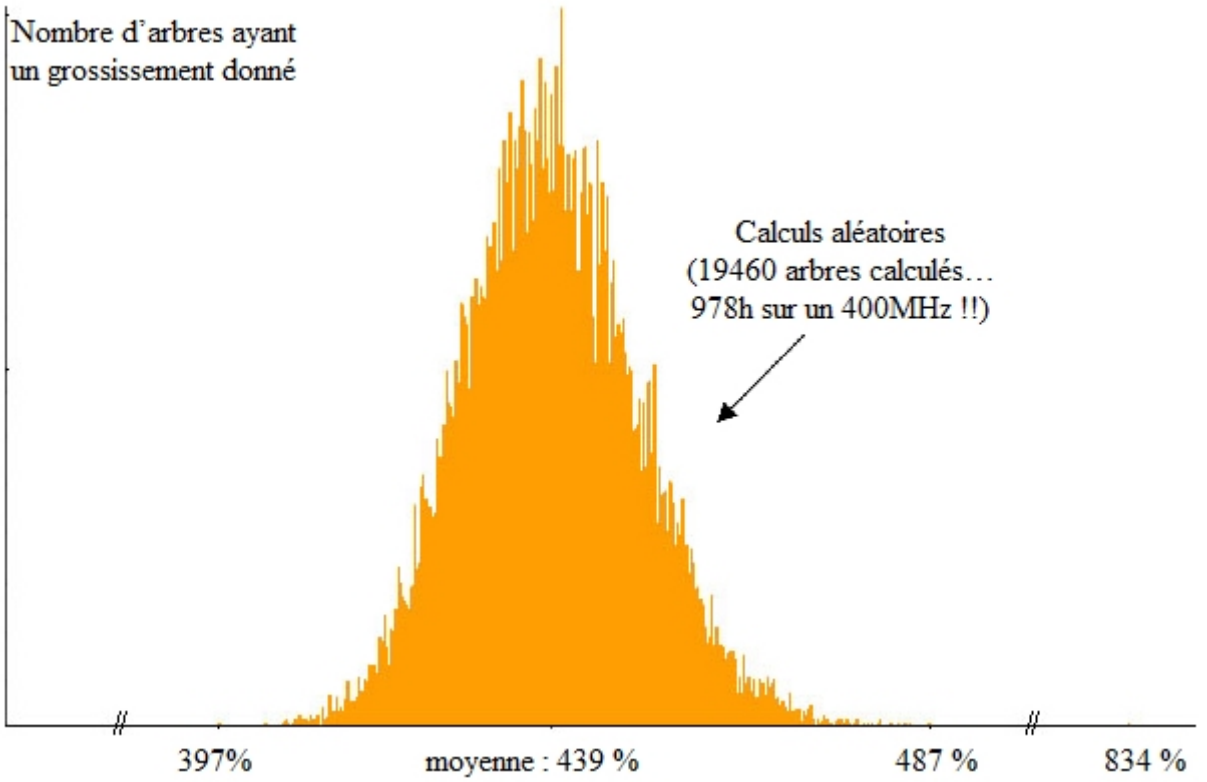



# Choix du plan de partitionnement

Afin d’optimiser la future lecture de l’arbre, le choix du plan de partitionnement est déterminant. Nous avons vu précédemment que certains plans de partitionnement (ou pivots) peuvent entraîner des découpes de faces donc un grossissement de l’arbre non négligeable. Le rendu final, en terme d’image affichée, ne dépend pas de l’arbre BSP. Par contre, le temps de calcul du rendu de l’image dépend du nombre de faces. Il n'existe pas d'algorithme permettant de choisir le plus petit arbre. Il est impossible de tous les calculer et de choisir le plus petit parce qu'il y en a beaucoup trop. Nous verrons aussi par la suite que l’arbre BSP le plus petit en terme de grossissement n’est pas forcément le plus adapté à un parcours optimisé pour le rendu pendant l’exécution du programme.

Par exemple, pour une bouteille de Klein de 3720 sommets, il existe environ 10^2160 arbres différents mais équivalents, équivalent dans le sens où l’image rendue est identique.

Afin de se rendre compte de la variation du grossissement d'un arbre l'autre pour le même objet, voici une étude statistique réalisée en calculant aléatoirement près de 20 000 arbres équivalents sur la bouteille de Klein présentée ci-dessus :




Extrait d'une étude réalisée par , professeur de MP*

Le grossissement est calculé de la manière suivante :

grossissement = (nombre final de faces dans l'arbre) / (nombre de faces initial dans le fichier)

Différentes solutions ont été proposées pour déterminer le choix des pivots mais seulement deux ont été retenues pour des implémentations efficaces d'algorithmes basés sur les arbres BSP.

## Choisir les pivots aléatoirement

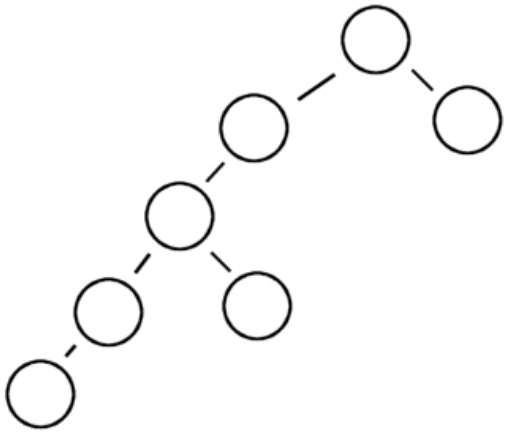
Le choix aléatoire des pivots est à double tranchant : on peut trouver un arbre acceptable (en terme de grossissement) comme un arbre disproportionné (grossissement > 800%). Dans ce cas, il faut calculer plusieurs arbres BSP avec une fonction de choix de pivot aléatoire, ce qui sous-entend un temps de calcul plus long (bien que le temps de compilation d'un arbre ne soit pas un facteur suffisamment déterminant si il permet d'obtenir un arbre optimisé pour le parcours) car plus d'arbres à calculer, mais surtout une occupation mémoire accrue, car il faut stocker les arbres pour ensuite les comparer.

## Choisir comme pivot la première face de la liste

Instinctivement, on aura tendance à penser que c'est un mauvais choix. Et on aura raison. En effet, les faces de la liste sont dans l'ordre dans lequel elles sont lues dans le fichier représentant l'environnement et il y a évidemment, beaucoup de faces consécutive seront coplanaires (les faces constituant un mur on été éditées les unes à la suite des autres, sûrement grâce à un « copier-coller »). Les plans de coupe seront donc choisi dans l'ordre dans lequel ils sont rencontrés, ce qui revient

## Choisir comme pivot le plan qui coupe le moins de faces

C'est première technique basé sur un principe logique : limiter au maximum le grossissement. On compare chaque face du monde avec toutes les autres et on choisit celle qui coupe le moins de faces. Cette méthode a une complexité en temps en $O(n^2)$ sur les faces de chaque nœud. En effet, pour la première subdivision, on compare toutes les faces entre elles pour choisir un pivot. A la suite de cela, une face est choisi et on divise l'espace en deux, une fois que les faces coplanaires sont associées au nœud courant (ici la racine) : les faces « avant » vers le fils droit et les faces « arrière » vers le fils gauche. On recommence ensuite la comparaison sur le fils droit et sur le fils gauche. Dans cette méthode, le temps de compilation devient vraiment très long pour finalement obtenir des résultats pas si satisfaisants. En pratique, il est logique que le premier pivot soit un plan qui contienne un mur (groupe de faces à afficher) limitrophe de l'environnement, puisque c'est lui qui coupe le moins de faces. Ce qui implique que la totalité de l'environnement va se trouver du même coté de ce mur. Il faut au moins quatre murs limitrophes pour délimiter un environnement (en fait, trois murs et un plancher, en supposant que cette environnement soit munis de règles physiques comme les collisions et la gravité) et on peut raisonnablement imaginer qu'ils seront les pivots des premiers niveaux de l'arbre construit. Ce qui va nous amener à une situation similaire à celle-ci :



On va obtenir un arbre qui ne sera pas équilibré. Le parcours d'un arbre tel que celui-ci équivaut (à peu de chose près) à parcourir une liste d'ensemble de faces coplanaires. On perd donc la possibilité d'éliminer le traitement d'une branche entière sur le résultat d'un test sur le nœud.

## Choisir le pivot en vue d'obtenir un arbre équilibré

Après avoir constaté que la méthode précédente ne permettait pas un parcours performant de l'arbre (voir sections « Initialisation du champ bounding box de chaque » et « Parcours de l'arbre »), des recherches ont été faites pour diviser l'environnement de manière à équilibrer l'arbre. On part du principe suivant : « même si cela doit engendrer plus de faces, lorsqu'on parcours l'arbre, on raisonne en ensemble de faces coplanaires ». On parcoure donc toutes les faces et on regarde combien de faces sont devant et derrière le plan porteur, puis on choisit celle qui engendre la différence minimale entre le nombre de faces avant et nombre de faces arrière. Cette méthode de choix de pivot est également en $O(n^2)$ sur les faces de chaque nœud. Le nombre de faces engendrées par les découpes n'étant que peu contrôlables, il est judicieux de stopper la récursivité lorsque l'arbre atteint une certaine hauteur, ou lorsque les feuilles contiennent un nombre de faces permettant d'être traitées plus vite à l'implémentation.

Par exemple, afin de minimiser les temps de rendu, OpenGL dispose dans son API de listes d'affichage. Une petite explication s'impose : lorsque le programme envoie une face à la carte graphique pour l'afficher, il le fait par le bus AGP. Donc chaque fois qu'une face est « envoyée » vers la carte graphique, on « paye » un passage par le bus AGP. Open GL offre la possibilité d'envoyer les faces par groupe grâce aux listes d'affichage, donc de ne faire qu'un trajet par le bus AGP pour un ensemble de faces. De plus, les faces contenues dans la liste d'affichage sont « compilées » afin d'obtenir un temps de calcul (position, éclairages, reflets, transparence, …) plus rapide sur ces faces. Mais l'utilisation d'une liste d'affichage n'améliore réellement les performances que si la liste contient au moins 200 faces.

En se basant sur ce que l'on sait maintenant sur les arbres BSP et les listes d'affichage, 200 à 400 faces est une longueur de liste associée à un nœud qui représente un bon compromis pour stopper la récursivité de la compilation de notre arbre.

## Conclusion sur le choix du pivot

Les méthodes de choix de pivot basés sur les comparaisons de faces sont en $O(n^2)$ sur le nombre de faces de chaque nœud. Les algorithmes sur les arbres BSP datent du début des années 80 et de multiples recherche sur les choix de pivot ont été réalisées. Aujourd'hui, on ne compare plus les faces uniquement en fonction du nombres de découpe qu'elles vont engendrer on en fonction de leur différence de faces « avant-arrière ». En effet, on attribut un score à chaque face de la liste qui dépend des ces deux paramètres et on garde comme pivot la face qui obtient le meilleur score.
Une méthode de calcul de score qui permet d'obtenir d'assez bons résultats est :

Score_Face_Courante = 2 x Nb_Faces_Coupees + abs (Nb_Faces_Devant – Nb_Faces_Derriere ) - Nb_Faces_Coplanaires

La face obtenant le score minimum est élue pour servir de pivot.

J'ai dit que cette méthode de calcul de score permettait d'obtenir d'assez bons résultat. En effet, les meilleurs méthodes pour compiler les arbres BSP et choisir leurs pivots ont été testées et implémentées par de grandes société de jeux vidéo (Id Software pour la saga des « Quake », Epic Games pour tous les « Unreal ») et il semble très difficile d'avoir des informations précises sur leurs travaux.

# Initialisation de la bounding box de chaque nœud

Le champ bounding box de chaque nœud va servir pour tester un nœud et éliminer éventuellement le traitement d'une branche de l'arbre.

On l'initialise de la manière suivante :
Si le nœud est une feuille, alors sa boite englobe toutes les faces de sa liste associée, Sinon, elle englobe les faces de sa liste associée et aussi les boites englobantes de ses fils.

Ce champ nous permettra de faire un test de visibilité pour les faces de la liste associée au nœud et pour les nœud fils pendant l'exécution.

Une fois les bounding box de chaque nœud calculée, on sauvegarde dans un fichier la structure arborescente avec les listes de faces et les bounding box associées à chaque nœud. Pour l'exécution, il n'est pas nécessaire de conserver les équations de plan.

# Parcours de l'arbre pendant l'exécution

Après le chargement du fichier représentant l'arborescence de l'environnement compilé, l'objectif est d'éliminer un maximum de faces qui ne sont pas dans le champ de vision en un minimum de temps. Pour ce faire, le principe du parcours de l'arbre est le suivant :

Si la bounding box du nœud courant est dans le champ de vision, alors on affiche la liste de faces associée au nœud courant, puis on réitère le test pour chacun de ses fils.

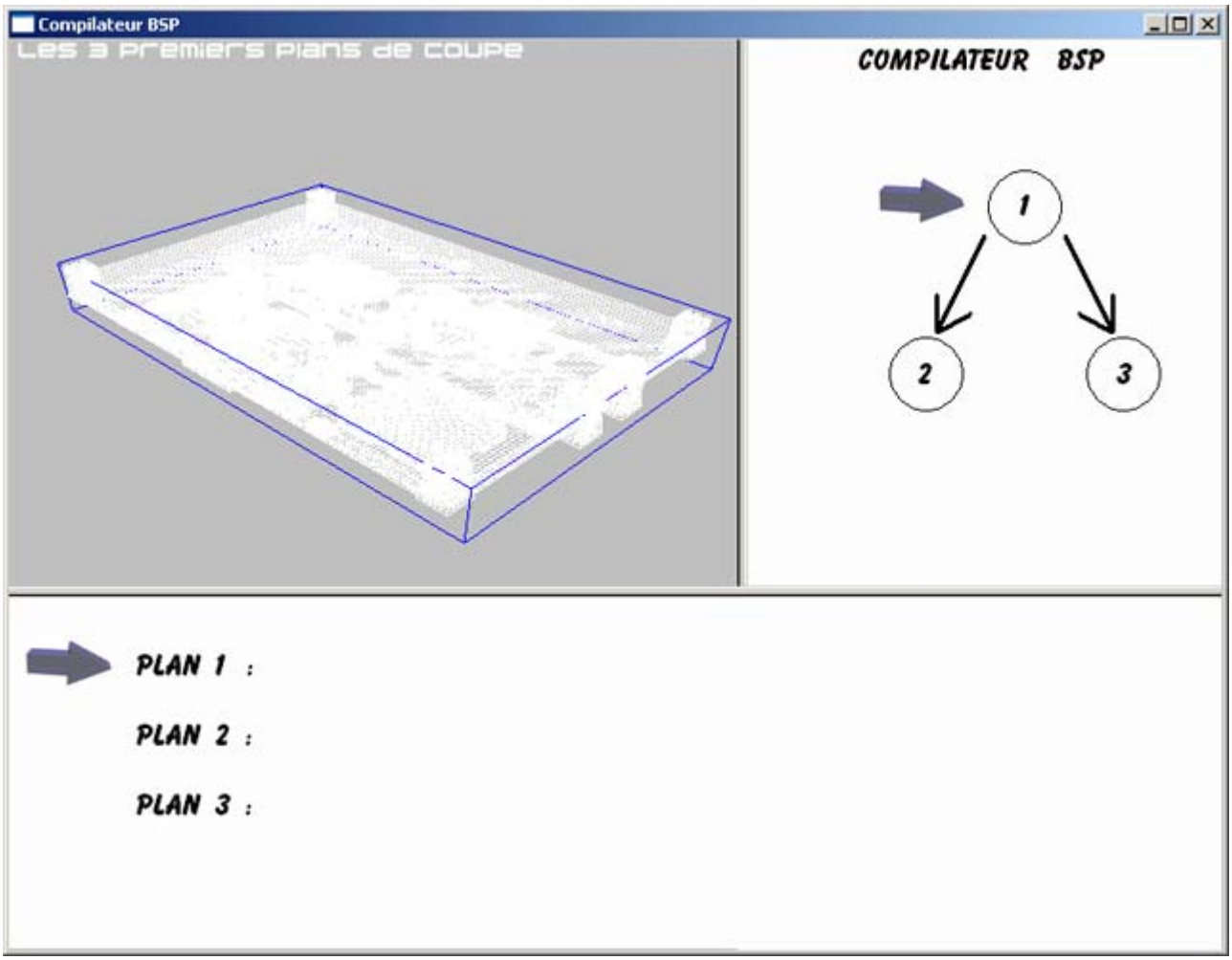
Si la bounding box du nœud courant n’est pas dans le champ de vision, on sait qu’aucune des faces de la liste associée au nœud courant n’est dans le champ de vision, mais on sait aussi qu’aucun de ses fils ne devra être affichés et on peut arrêter la récursivité.

Une branche entière de l’arbre peut être occultée du traitement sur un simple test de visibilité, ce qui met en valeur l’intérêt de chercher à obtenir un arbre équilibré à la compilation.

# Programmes implémentés

## Compilateur d'arbres BSP

L'objectif était de programmer une interface graphique au compilateur d'arbres BSP avec la possibilité de voir les 3 premiers plans de coupe (la racine de l'arbre et ses deux fils) avec la possibilité d'imposer les équations de ces plans pour la compilation si des plans de coupe optimum sont évidents pour l'utilisateur. Malheureusement, les problèmes rencontrés lors de l'implémentation furent plus difficiles à résoudre que prévu et je n'ai pu aboutir à un programme fini. En effet, une des principales difficultés par rapport aux quadtrees ou aux octrees est de gérer les plans de coupe « presque coplanaires », c’est à dire coplanaires dans la réalité mais à cause de l’approximation des nombres réels, ils ne sont pas coplanaires par l’ordinateur, Avant de gérer cela, ces plans presque coplanaires engendraient des découpes multiples en « lamelles ultra fines », ce qui générait un nombre considérable de faces supplémentaires. L’implémentation d’arbres BSP doit se faire avec une extrême minutie et la détection d’erreurs devient très difficile.




## Moteur 3D utilisant l'arbre BSP compilé

L'algorithme du parcours de l'arbre me semble opérationnel mais je n'ai pas eu la possibilité de le tester car je n'ai pas pu terminer le compilateur.

## Conclusion sur l'utilisation des arbres BSP

Les arbres BSP offrent de très bonnes performances dans la simulation d'un déplacement dans un environnement indoor composé d'une multitude de pièces. Ces performances se mesurent par la fluidité d'affichage (nombre d'images par seconde) ou bien par le pourcentage de polygones de l'environnement affiché. Ces deux métriques sont en étroite relation dans la mesure ou le taux de rafraîchissement dépend directement du nombre de polygones affichés pour une même configuration matérielle.

Dans un environnement extérieur, ils sont nettement moins adaptés. En effet, ces environnements disposent également de plans de coupe, mais les plans de coupe sont calculés en fonction des meilleurs scores obtenus par les plans porteurs des faces de l'environnement. Et les meilleurs plans de coupe dans les environnements extérieurs ne sont pas bons. Les grandes étendues planes avec, par

exemple, des arbres et quelques bâtiments obtiennent de très mauvais scores car ce type de configuration n'autorise qu'un volume de vision profond, ce qui ne permet que l'occlusion des nœuds se trouvant derrière la caméra.

L'efficacité des arbres BSP est à mettre en parallèle avec la difficulté de leur implémentation. En partant de peu de connaissances sur le sujet, il a fallu étudier les structures de données, réaliser un environnement vaste afin de voir les performances de chaque algorithme étudié, apprendre à utiliser un librairie telle qu'Open GL, implémenter un compilateur d'arbres BSP et réaliser un moteur basé sur ces mêmes arbres BSP en un temps excessivement restreint pour un tel volume de travail.

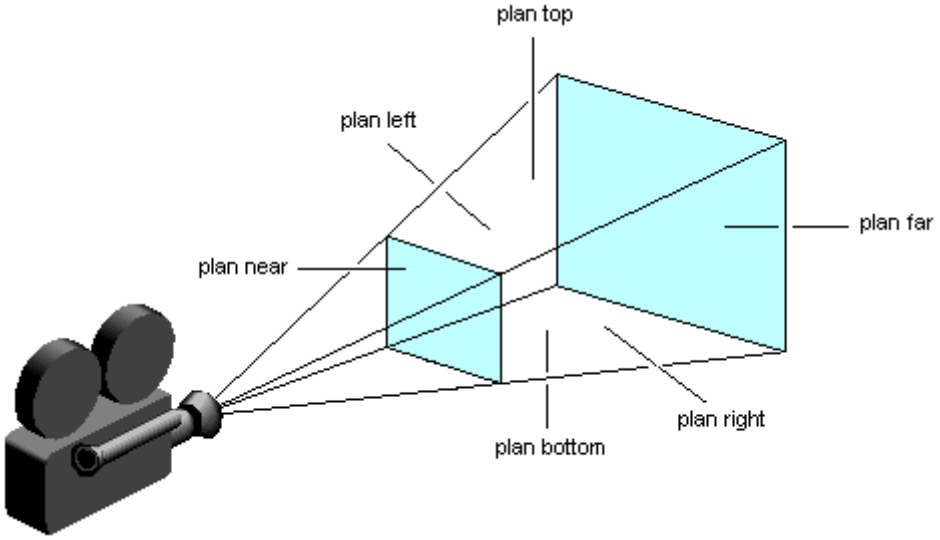
# Les Portails

## Qu'est ce qu'un portail?

Les algorithmes à base de portail visent à découper l'univers en secteurs.
Ce qui relie un secteur à un autre est le fait que si on se trouve dans un secteur, et si on peut voir une partie d'un autre secteur, alors ils sont voisins.

Et la partie qui les relie est "un portail". En fait, cette structure de données permet lorsqu'on se déplace de ne considérer que les polygones appartenant au secteur courant plus les polygones des secteurs voisins.

Pour calculer ces "secteurs" et déterminer leurs relations de voisinage, c'est assez simple :
On regarde si depuis un secteur on peut "voir" des éléments qui ne sont pas dans le secteur courant.
La difficulté consiste dans cette phase, justement à déterminer ce qui est visible depuis un secteur donné. On se base sur le champs de vision.




Pour entrer dans les détails de l'algorithme on va prendre quelques exemples
les secteurs sont des pièces composés de polygones. Comprenez par pièce une salle en 3D comme un salon ou une chambre et contenant des portes qui les relient.

Comme une pièce de la vie de tous les jours
Il faut un fichier annexe contenant des informations sur les pièces (hauteur, largeur, position)
et des informations sur les portails.Un portail est un polygone apartenant simultanément à 2 pièces, dans notre cas 2 pointeurs sur des pièces.
Il est donc nécessaire de loader un fichier spécial au lancement du programme pour compiler le monde
Il parait que certaines entreprises de jeux videos travailleraient sur un algo permettant de se passer
de ce fichier, ainsi le monde serait compilé a la volée, info ou intox?

# Présentation de l'algorithme

Après réflexion, nous avons décider de vous présenter l'algorithme sur les portails à l'aide d'un soft 2D, en vue de dessus.
La structure principale de l'algorithme basé sur les portails est la classe Frustum
Il n'est pas aisé de "montrer" des transformations de frustum en 3D. En effet cela s'apparente
à des effets spéciaux visuels (motion blur, zoom , etc). Les librairies OpenGL et Directx, possèdent
une méthode d'extraction du frustum. Cette méthode consiste à récupérer les matrices Modelview et Projection(lien)

Mais nous n'avons pas voulu nous en servir et avons préféré tout réécrire from scratch. Ainsi la classe caméra, qui lui est étroitement liée, est inspirée de celle utilisée dans le moteur 3D, mais possède ses propres fonctionnalités.
De plus, l'algo fonctionne avec un fichier annexe conservant les informations des pièces, en 3D il aurait fallu un éditeur de pièces (cf Annexe en construction) pour pouvoir avoir une scène complète.
De plus, les fonctionnalités intéressantes comme le clipping n'auraient pas été visibles. En effet, en 2D on peut tout de même afficher le monde entier, et surligner ce qui serait effectivement
envoyé a la carte 3D.
Enfin, je tiens à préciser que tous les calculs vectoriels et analytiques sont faits en 3D en se servant
de Points ayant une coordonnée en z égale à zéro.

# Notions mises en jeux dans l'algorithme

**Les notions primordiales sont :**
Une pièce **est un secteur du monde virtuel**
Un portail **est un polygone spécial du monde virtuel, il relie deux pièces(p1, p2) entre elles**
La récurrence **est le principe sur lequel est basé l'algo.**
Le frustum **est un volume symbolisant le champs de la caméra**
Le clipping **est une technique qui permet de récupérer la partie visible d'un polygone**

# Les pièces, les portails et la récurrence

## le prédicat de base et la récurrence

La classe pièce est un secteur du monde, le monde est formé d'un tableau dynamique de pièces,
mais on s'en sert comme une liste pour les fonctions de recherche d'élément. C'est un volume
de l'espace qui représente un pièce de la vie de tous les jours (salon, chambre,...). Deux pièces
peuvent être reliées entre elles par un portail,

c'est un polygone spécial.
Aparté: Ici le terme "voir" signifie "être dans le frustum" que ce soit en 2D ou en 3D c'est la même chose.
Imaginons un pièce A et une pièce B, mitoyennes, reliées entre elles par un portail p.(figure 1)
"Si je suis dans la pièce A et que je vois le portail p alors je vois la pièce B."
Ceci est le prédicat de base de l'algorithme.
La figure 2, apporte un problème intéressant. En effet, si "je me trouve dans la pièce A et que
je regarde vers la droite". " Je vois le portail 1, donc je vois la pièce B ". Mais " je vois aussi
la pièce C via le portail 2 ". Ceci n'est pas pris en compte dans le prédicat, c'est ici qu'intervient
la notion de récurrence. Pour chaque pièce visible "je vais visiter tous les portails visibles associés
à cette pièce et itérer sur le prédicat". Ainsi pour le cas de la figure 2, l'algorithme va permettre de considérer que la
pièce C, est visible de la pièce A.

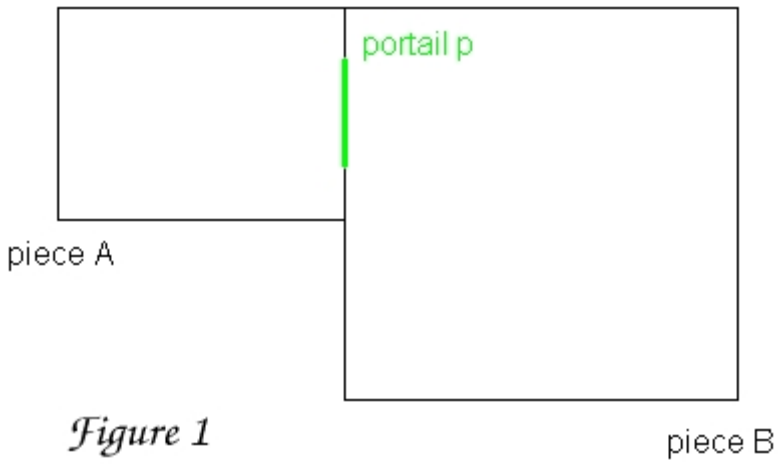



*Figure 1*

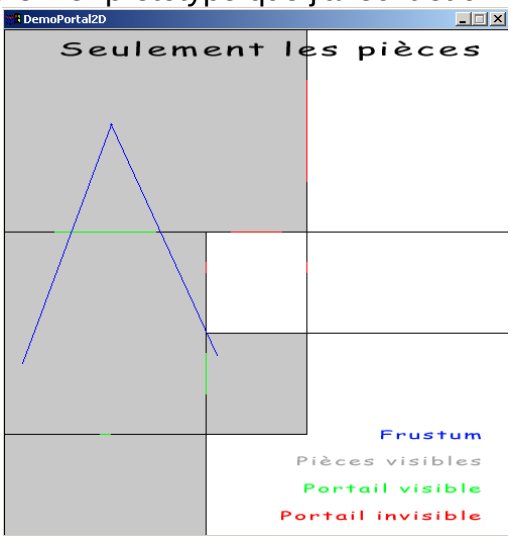
*Figure 2*

Mais on a alors problème de cas d'arrêt !!

Le processus, décrit ci-dessus est une boucle sans fin, car "de la pièce A je vois le portail 1".
Donc "je vois la pièce B et je considère alors la pièce B". "Dans la pièce B, je vois le portail 1,
donc je vois la pièce A... et ainsi de suite". Il faut donc marquer les pièces et les portails déjà visités
Ce processus s'apparente donc a un parcours de graphe non orienté en profondeur, à partir d'un nœud spécial.

## Comment représenter une pièce et un portail ?

L'algorithme marche donc théoriquement, il faut maintenant, l'implémenter. A partir de maintenant, je vais entrer dans
les détails de mon code.
J'ai eu besoin d'une structure de pièce, de portail et de frustum
Voici un screenShot du premier prototype que j'ai écrit et un lien



Pour le moteur 2D et le frustum je me suis servi de la démo de Jean-François sur les quadtrees en 2D.
Le frustum avait un petit bug non bloquant. Il a fallu que j'écrive un fichier texte dans lequel je conserve
les infos sur les pièces et les portails dont voici la syntaxe

NB_PIECES nbpieces

NB_PORTAILS nbPortails

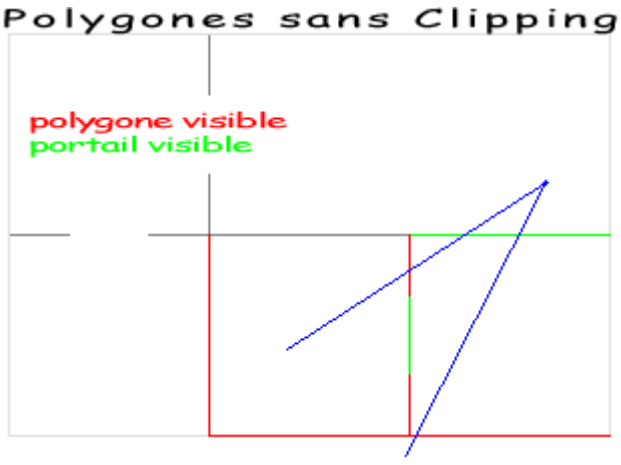
piece indice sommet1X sommet1Y sommet2X sommet2Y

portail indice sommet1X sommet1Y sommet2X sommet2Y pieceA pieceB

nbpieces est le nombre de pièces présentent dans le monde.
nbPortail est le nombre de portails présents dans le monde.
Une pièce est caractérisée par un indice et 2 sommets, qui sont les coins opposés d'un rectangle,
ou d'un parallélépipède dans le cas de la 3D.
Un portail est caractérisé par un indice, 2 sommets (cela représente un segment) et les indice des 2 pièces qu'il relie.
Il n'y a pas d'analyse sémantique, donc je pars du principe que le fichier texte
est bien formé. C'est à dire que le nombre de ligne commençant par pièce est égal à NB_PIECES.
Idem pour NB_PORTAILS. De plus, les pièces doivent être collées et les portails bien placés.
J'en suis arrivé à la conclusion qu'il fallait un Editeur. (cf Annexe)

## Polygones et portails

En introduction, j'ai dit qu'un portail était un polygone spécial.
Or il n'y avait pas de stucture de donnée polygone dans la démo précédente.
J'ai donc repris mes classes, réécrit une nouvelle syntaxe et fait une nouvelle démo.
Dont voici un screenShot et un lien vers l'exécutable.




Ici, le but était de se passer de l'affichage de la classe Piece (le carré gris de la 1ere Démo) et de n'afficher que les polygones la formant: les murs. Deux nouvelles structures de données apparaissent :la classe Polygone et la classe Sommet,
qui lui est associée. Il a alors fallu que je réécrive ma syntaxe.

Voici la nouvelle et dernière syntaxe

```
sommet        indice x y
polygone      indice indiceSommet1 indiceSommet2
pièce         indice indiceSommet1 indiceSommet2
linkPolygone  indicePolygone indicePièce
portail       indicePolygone indicePièce1 indicePièce2
```

Comme on peut le remarquer il n'y a plus écrit ni le nombre de pièces, ni le nombre de portails. C'est devenu inutile. Je me suis inspiré de la syntaxe des .obj. Donc la première chose à faire est de charger tous les sommets. Puis de créer les polygones, par exemple polygone 3 2 9 signifie que le polygone d'indice 3 à comme sommets les points d'indice 2 et 9. Ensuite on écrit les informations sur les pièces, de la même manière que précédemment,
sauf que les coins sont des sommets chargés. Après cela, les polygones sont liés aux pièces.
On peut, et doit dans le cas des portails, linker un polygone à plusieurs pièces.
Enfin, on fait un setPortail() sur certains polygones du monde. portail 2 3 4
signifie que le polygone d'indice 2 est un portail reliant la pièce 3 à la pièce 4.
L'algorithme en lui même reste inchangé on lance une récurrence et pour tous les polygones de la pièce. Si il est visible on l'affiche. La démo marchait bien mais il me restait encore 2 choses à régler.
Le clipping et la diminution de frustum.

# Visiblité clipping vs Bouding Box

Depuis le début de ma partie du rapport, je me sers du terme "voir";
"si je vois le portail p alors je vois la pièce B".
"Si je vois le mur i alors je l'affiche."

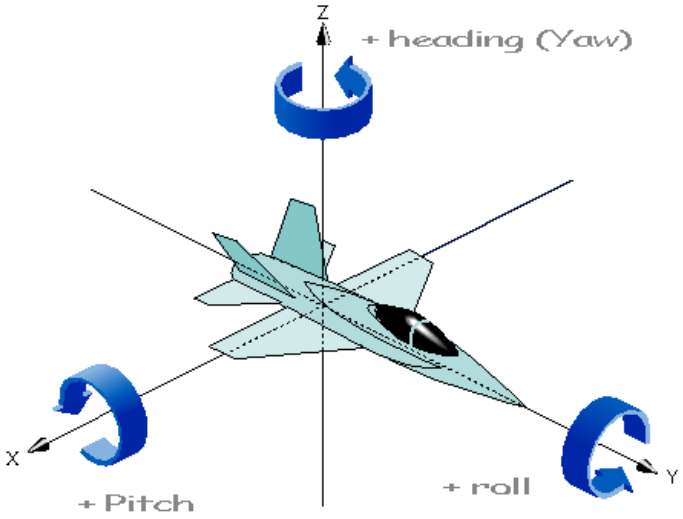
Et j'ai précisé qu'un polygone était visible si il était dans le frustum. Il faut donc que j'explique ce test de visibilité. Pour expliquer cette notion, je vous propose des rappels
de ce qu'est le frustum et ma manière de le créer. Si vous avez déjà lu le petit article de
Jean-François Fourmond à ce propos. Vous trouverez sûrement des redites, mais je pense être plus précis au niveau géométrique et analytique.
Nous vous conseillons donc de lire les deux pour avoir une vue d'ensemble et de commencer par
celui de JF pour les schémas

## les classes Frustum et Caméra

Un frustum est lié à une caméra, nous allons nous intéresser à celle ci en premier lieu.

## La classe Caméra

la classe caméra représente l'avatar que l'utilisateur déplace dans le monde virtuel elle est caractérisée par une vitesse, une **position** et 3 angles: **heading(Yaw), pitch et roll** (cf figure ci dessous).




Ces termes sont inspirés des termes de l'aviation. Ainsi quand on connaît ces 4 données, on peut
en déduire un droite orientée de l'espace, définissant vers "où pointe la caméra", (cf fonction lookAt de glut)
Cette droite peut etre représentée par un Vecteur et un Point, par exemple.
Une fois ce Vecteur connu on peut construire le frustum associé à la caméra,
c'est à dire une portion de l'espace devant la caméra représentant
le volume visible par l'avatar

Dans un fps "classique", le joueur fait varier 2 de ses angles, le heading et le pitch.
Dans les fps, "nouvelle génération": Battlefield 1942, BF Vietnam, Soldner.
Le joueur peut se servir de véhicules aériens et dans ce cas les 3 angles sont Mais de façon naturelle: avion, hélicoptère Moi en 2D, je n'ai besoin que du heading.
Si j'avais voulu créer le frustum en 3D, il m'aurait suffit de rajouter l'angle pitch de la caméra dans les calculs d'extraction des 8 points.
De plus mon frustum étant une coupe d'un Frustum 3D, il possède seulement
4 plans que j'ai appelés front, back, right et left.

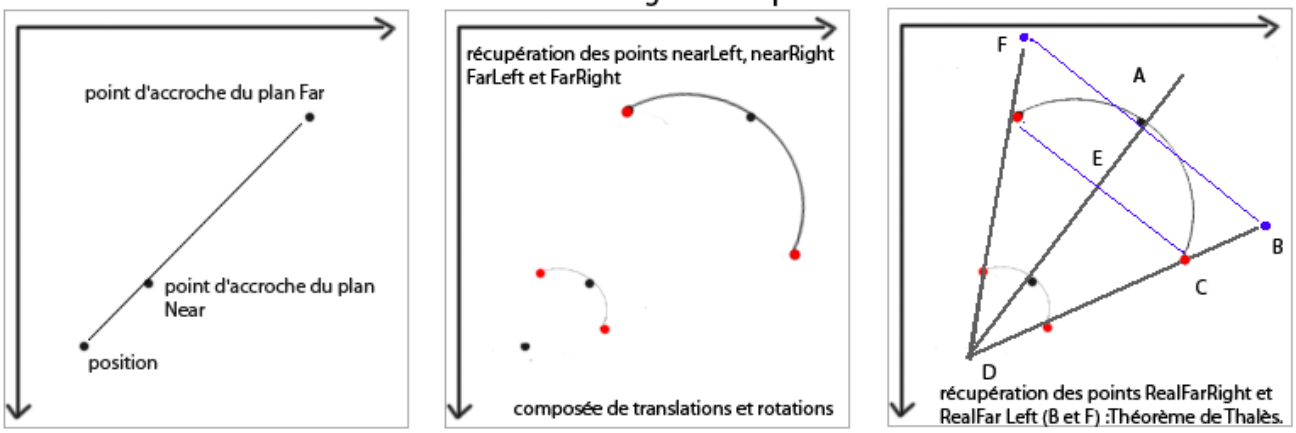
# Revenons au Frustum

La librairie OPENGL possède des méthodes d'extraction du frustum.
Or je suis en 2D et je ne me sers pas de cette librairie, il a fallu que je réécrive les méthodes de création from scratch
Ici interviennent des notions de géométrie analytique dans l'espace
Ce calcul se fait en 2 passes.

## Première passe: extraction géométrique de 4 points du plan qui sont les points d'accroche du frustum

Un dessin valant mieux que de long discours je vous présente donc ma méthode de construction du frustum.
De façon géométrique.

Méthode de construction géométrique du frustum




je vous invite à lire les sources de la classe Frustum pour plus de précisions.

## Deuxième passe: calcul des équations des 4 plans : Petits rappels Mathématiques: les classes Plan, Vecteurs et Point

**Un Point** est caractérisé par 3 coordonnées (Réelles) x,y et z ce sont donc des float en C++
**Un Vecteur** est caractérisé par 3 coordonnées (Réelles) x,y et z ce sont donc des float en C++
Pour définir un Vecteur v (le constructeur en C++) il faut 2 Points: origine (o) et extrémité (e)
On a alors:

$$v.x = e.x - o.x$$
$$v.y = e.y - o.y$$
$$v.z = e.z - o.z$$

L'équation d'un **Plan** dans l'espace est alpha * x + beta * y + gamma * z = delta.
Ce qui signifie qu'un Point, de coordonnées (x,y,z), appartenant à ce plan vérifie cette équation
Pour trouver l'équation d'un plan il existe plusieurs méthodes(Analytique, géométrique,?trigonométrique?).
Je vous expose ici celle dont je me suis servie.

Elle s'apparente à un problème mathématique et je l'ai traité comme tel:

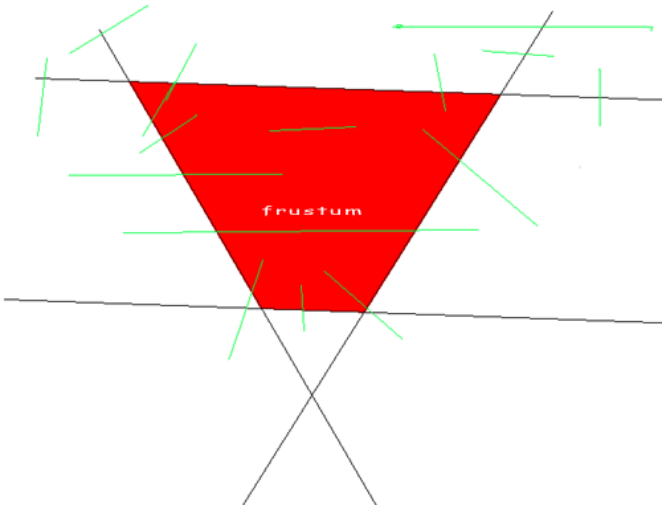
Si on possède 1 Point de l'espace et un vecteur normal à ce plan, on peut en déduire l'équation du plan
Pour trouver le vecteur normal à un plan, on peut de servir du produit vectoriel de 2 Vecteurs appartenant à ce plan.
Pour trouver 2 Vecteurs du Plan, il suffit d'avoir 3 Points A,B,C appartenant au Plan .
Le calcul de la normale se fait donc par Vecteur Normal = Vecteur[A,B] * Vecteur[A,C] (où * représente le produit vectoriel)
Je ne rentre pas dans les détails du calcul, mais invite le lecteur à lire mes sources pour les calculs
mathématiques
(classe Vecteur)

Une fois le vecteur normal trouvé, il suffit de le **normaliser**.
Normaliser un vecteur,non nul, consiste à diviser chacune de ses coordonnée(s) par **la norme** de ce Vecteur. La Norme d'un Vecteur étant la distance entre ses points "extrémité" et "origine"(cf sources pour le calcul).
Une fois le Vecteur Normal trouvé et le Point (M) d'ancrage du Plan définit,
nous avons toutes les informations en main pour trouver l'équation du Plan.

Pour cela il faut se servir des propriétés du produit scalaire et de l'appartenance du Point M au Plan.
Ainsi Vecteur NormalVecteur Normal . Vecteur[O,M] = 0 (où . représente le produit scalaire)
et
M appartient au Plan
on en déduit alors les paramètres alpha, beta, gamma et delta du Plan(cf constructeur de la classe Plan).

## Visibilité

Une fois tous les plans calculés, on peut alors faire des tests de visibilité.
Dans le cas de la dernière démo présenté le test de visibilité s'apparentait à un test de Bouding Box.
Ainsi un polygone était considéré comme visible, si au moins un de ces sommets était présent dans le frustum.
Les explications de ce cas de figure sont faites dans l'article de Jean-François.
Or, si les deux Sommets étaient à l'extérieur du frustum et que le polygone coupait le frustum comme dans le cas du grand segment du milieu sur la figure ci dessous.
Le polygone était considéré comme non visible alors qu'il l'était effectivement.




La solution à ce problème a été de clippé les polygones sur le frustum.
"Clippé" signifie récupérer la partie du polygone se trouvant dans le frustum.
Pour cela je me suis servi de l'algorithme de Sutherland-Hodgeman.
Voici en pseudo-code C++ la manière dont je l'ai implémenté.
C'est une version épurée d'un post que j'ai placé dans le wiki donc il se peut qu'il y ai des encore des erreurs.

Appel

```
  bool clipPolygone(listePolygone[i],new Polygone())
```

Variables

```
  ArreteCourante: c'est un segment
  out in et in out selon le type de repère(main gauche ou main droite).
  point inter point d'intersection de l'ArreteCourante avec le plan si il a lieu d'etre.
```

Pseudo Code

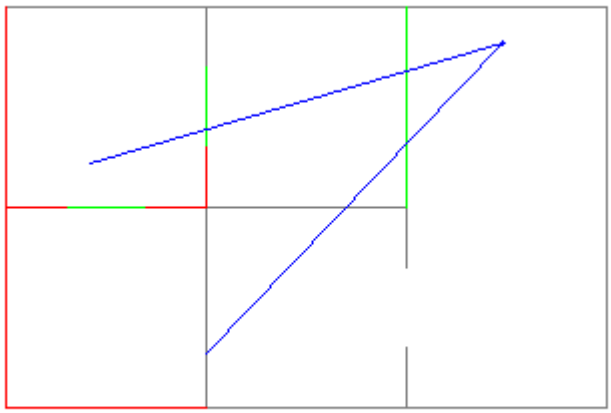
```
bool clip(Polygone *p, Polygone *pRes)
    //les deux points "courants"
    Point extermité;
    Point origine;
    bool clip=true;
    int  cptSegmentsClipés;
    <Sommet > listeSommets = vide;
    <Sommet > listeSrc = p->toSommets // crée une liste de sommets  qui definissent le polygone
    Pour toutes les arretes faire {   //on parcourt toutes les arretes il faut un pointeur sur
l'arreteCourante
        extremité =  arreteCourante.S1;
        origine   =  arreteCourante.S2;
        Pour chaque plan du volume faire {
          float distanceFromOrigineToplan /*d1*/ = plan.distance(origine);
          float distanceFromOrigineToplan /*d2*/ = plan.distance(extremité);
```

```
        Selon le signe de d1 et d2 il y a 4 cas possibles
            out    : clip=false;//ce segment n'est pas dans le volume c'est sur
            in     : ;
            in out : pointInter = Point();
                     pointInter=  calculerIntersection(extremite,origine,plancourant)//
                     extremité=pointInter;
            out in : pointInter =Point();
                     pointInter=  calculerIntersection(extremite,origine,plancourant)//
                     origine = point inter
      }
  }
  return clip;
}
```

Ainsi dans la dernière version de ma démonstration (lien) ne sont affichés que les parties visibles des polygones.

## Le recalcul du frustum

Un problème reste encore à régler, illustrer par ce screenshot.



bug occlusion

Ici, il y a un portail considéré comme visible alors qu'il ne l'est pas.
Le frustum doit être recalculer dans le cas où l'embrasure du portail est petite.
C'est le même principe que le "lancer de rayon" (ray casting) et est loin d'être un problème trivial.
Dans la démonstration v0.9.5 , en mode nightmare, j'ai commencer à implémenter une piste pour régler ce problème.
J'extrait le point médian d'un portail quand il est clippé par le frustum.

# L'algorithme

Voici un squelette de l'algorithme.
C'est un résumé de ce que l'implémentation faite dans la classe Monde.
L'algorithme en lui même est divisé en 3 fonctions
**affiche(...)**
**affichePiece(...)**
**affichePortail(...)**
affichePiece et affichePortail s'appellent l'une, l'autre;
ces appels se font en position terminale(cf Annexe)
Au moment où la carte graphique demande une nouvelle frame/trame à afficher,
la fonction affiche() est appelée.
La première action de cette fonction est de recalculer le frustum selon la position
et les angles de la caméra, ici le Heading seulement.

La seconde est de trouver la pièce courante: "où se trouve la caméra?"
Une fois cette pièce connue, la fonction affichePiece(pieceCourante) est appelée.
La première action entreprise dans cette fonction est de marquer la pièce comme
ayant été visitée
La seconde est de clipper tous les polygones sur le frustum.
Cela revient à trouver si ils sont visibles;
et plus précisément à extraire leur sous partie visible, si cette partie
est nulle le polygone n'est pas visible. Si ce polygone est "normal",
un mur par exemple, on l'envoi à la carte graphique(faire des paquets et non un par
un).
Si ce polygone est un portail, non marqué, la pièce voisine associée à
ce portail est dans le champs de vision.
Donc certains murs de cette pièce sont potentiellement visibles.
La fonction affichePortail(portailCourant) est alors appelée.
Cette fonction, marque le portail comme visible puis
appelle la fonction affichePiece() sur les 2 pièces reliées par le portail.
Cet algorithme est donc un parcourt de graphe en profondeur, où les nœuds sont
Les nœuds étant les pièces, les arêtes étant les portails.

## Ray Casting implémentation

je pense que je modifierai ma fonction affichepiece() de cette manière(src:
www.flipcode.com)

```
Frustum fr;
   function renderSector (sector)
   {  for (each polygon in current sector)
      {  if (!facing (polygon)) continue;
         if (clip (polygon, fr) == NULL) continue;
         if (polygon != PORTAL) markVisible;
         if (polygon == PORTAL)
         {  Push fr;
            fr = adjustFrustum (fr, polygon);
```

```
            renderSector (polygon.sector)
            Pop fr;
        }
    }
}
```

# Optimisations et réflexions

## Calcul matriciel

Ma classe Matrice possède 3 lignes et 3 colonnes, les coordonnées ne sont donc pas homogènes.
Ce qui me permet de gagner des calculs.

## Recherche de pièce courante

Je parcours toutes les pièces, avec une boucle for, pour trouver la pièce courante

## Meshes

J'ai écrit cette classe, elle existe dans mon code. Mais je ne l'ai pas utilisé.
Elle servirait à englober un groupe de polygones afin de diminuer les tests de clipping.

## Récursion Terminale vs Iteration

Il y a 2 fonctions s'appelant l'une l'autre dans cette algorithme. Elles sont toutes les deux positionnées en position terminale.

Donc on peut dire que c'est de l'itératif (la hauteur de pile reste inchangée).

Je suis aller voir Mr Roy (Professeur à l'UNSA) afin qu'il m'aide à me faire un jugement.
Or après des tests avec un de ses collègues nous nous sommes rendu compte qu'un programme,
compilé avec g++ et ayant ces 2 fonctions.

```
void f1{
  f2();
}
void f2(){
  f2();
}
int main(){
 f1();
}
```

s'arrêtait tout seul au bout d'un temps relativement court.
Ainsi la gestion de la récursivité du compilateur g++ est-elle floue.
N'ayant pas (encore) d'éditeur de pièces, je n'ai pas testé mon programme avec un grand nombre de salles.
Et j'ai du mal a avoir un point de vue, quant à "son comportement avec un grand nombre de pièces".

# Conclusion et Annexes

L'algorithme d'affichage des portails est donc un parcours de graphe non-orienté.
Ce graphe est représenté par une structure physique de pièces et de portails.
Les informations sur cette structure étant contenues dans un fichier annexes.
La recherche de la pièce courante est linéaire par rapport au nombre de pièce du monde
et le parcours de graphe est en grand to de (nombre pièces + nombre de portails).
Ce projet m'a demandé des connaissances aussi bien mathématiques(niveau Première-Terminale S)
qu'informatiques (récurrence)
Je tiens à préciser que la méthode de découpage d'un monde virtuel grâce aux portails est inusitée, voire obsolète.
En effet, on ne trouve que le jeu "Descent" (Parallax software) qui soit basé dessus et il date de plus de
10 ans. Ce qui représente beaucoup dans le monde du jeu vidéo. Les 3 axes de rotation de la caméra étaient libres, c'est la raison pour laquelle ce produit était malheureusement peu jouable.
Mais c'est en partie à cause du monopole du moteur de Quake qui été repris par tous les FPS depuis les dernières années.
Pour conclure je pense qu'un jeu 3D basé sur les portails n'apporterait rien a l'univers du jeu vidéo.
Mais le principe d'occlusion culling et de frustum 2D peut être repris sur des plateformes de types téléphone portable, PDA en réseau par exemple.

Je vous ai présenté ici, tout le travail concernant l'implémentation des algorithmes des portails. Or je n'ai pas fait que ça pendant ce TER, j'ai aussi aider Xavier à concevoir le Monde. De plus j'ai eu plusieurs idées qui me sont venues, lors de l'implémentation et je vous les présente
dans les annexes du Wiki. Enfin je voudrais finir par dire que le C++ n'est peut être pas l'outil idéal pour faire des maths et un langage comme Mapple me semble plus adapté.

# Synthèse sur l'étude de ces algorithmes

| | Quadtrees | Octrees | Arbre BSP | Portails |
|---|---|---|---|---|
| Environnement optimal | extérieur plat | extérieur | intérieur | intérieur |
| Construction de l'arbre | à l'exécution | à l'exécution | précompilation | à l'exécution ( graphe ) |
| Type de partitionnement | perpendiculaire ( alignés aux axes du monde) | perpendiculaire | non perpendiculaire (déterminé par des plans de coupe | par pièces |
| Type de division | division de régions carrées de plus en plus petites | division de régions cubiques | division des surface devant et derrière le plan | Décomposition du niveau en pièces |
| Possibilité d'utilisé avec les collisions | oui | oui | oui | oui (grâce aux définitions de pièces) |
| Utilisé avec le frustum culling | oui | oui | oui | oui |
| Complexité de construction | O(p^4) p = profondeur | O(p^8) | O( n^² ) sur les face de chaque nœud | Chargement des pièces avec celui du monde |
| Complexité de recherche | O(p log p) | O(p log p) | O( log n ) sur un arbre équilibré | O( n ) sur les pièces |
| Algorithme statique | oui | oui | oui | oui |
| Algorithme dynamique | non | non | non | possible, mais complexe |

Nous allons ici comparer les algorithmes étudiés, montrer leurs avantages, inconvénients, en fonction de l'environnement 3D dans lequel on se trouve, des difficultés d'implémentation, etc...

- Environnements

Quand on est à l'intérieur d'un bâtiment, les quadtrees et les octrees ne sont pas recommandés. En effet, si on est face à un mur qui occupe tout l'écran, on affichera quand même tous les polygones visibles qui sont derrière et qui sont moins loin que le plan far du frustum. Avec les arbres BSP, on peut éviter ce problème, ainsi qu'avec les portails, où l'on n'affiche que ce qu'on voit. Les quadtrees sont encore moins performants que les octrees si on a des étages, parce qu'on va afficher toute la hauteur d'un bâtiment

En revanche, les portails, qui sont basés sur des pièces, ne peuvent pas être utilisés dans des environnements extérieurs.
Si on est dans un monde sans pièces et donc sans murs, tout sera affiché avec les portails, même les polygones qui sont derrière la caméra. Il est préférable d'utiliser les quadtrees ou les octrees pour ce genre d'environnements. En effet, on souhaitera afficher les polygones aussi loin que possible et les quadtrees et octrees permettent de ne presque pas afficher de polygones qui sont hors du champ de vision.
Les quadtrees et octrees sont donc utilisés dans les environnements en extérieur, mais pas dans les mêmes cas. Par exemple si on est dans une ville, on a des immeubles de forme rectangulaire posés sur le plan du sol. Ca ne servirait à rien de les découper en cubes, ça ne ferait qu'augmenter le nombre de polygones affichés. Par contre si on a des montagnes constituées d'un certain nombre de polygones, il vaut mieux avoir des octrees car on peut ainsi afficher une partie des polygones de la montagne seulement selon qu'on regarde en bas ou à son sommet.
Avec les quadtrees, on affiche tous les polygones contenus dans un rectangle, quelque soit leur hauteur. On risque donc d'en afficher qui sont au dessus ou en dessous du champ de vision. Les arbres BSP ne sont pas efficaces avec les environnements extérieurs parce qu'on a pas de plan de coupe efficace.

- Difficulté d'implémentation

Les quadtrees et les octrees sont les plus simples à implémenter. Les arbres BSP et les portails sont bien plus complexes à mettre en oeuvre. Les quadtrees et octrees peuvent être créés pendant l'exécution mais les arbres BSP nécessitent une phase de pré-compilation de l'environnement et les portails ont besoin d'informations supplémentaires correspondant aux pièces.

- Utilisation mémoire

Les arbres BSP utilisent de la place sur le disque en raison du stockage de l'environnement compilé. La quantité de mémoire occupée pour stocker un arbre BSP après chargement est également plus importante qu'avec un quadtree ou un octree. En effet, l'algorithme de compilation étant essentiellement basé sur des plans de coupe, cela engendre un grossissement non négligeable du nombre de faces composant l'environnement. Les octrees prennent plus de place que les quadtrees car on a plus de subdivisions de l'espace : un octree a 8 fils, contre 4 pour un quadtree.

# Conclusion

**Adéquation avec le cahier des charges** : Nous avions prévu de faire une étude détaillée de chacun des algorithmes et de réaliser pour chacun d'eux un programme d'illustration pédagogique mettant en valeur leurs caractéristiques. Ceci a été fait pour les quadtrees et les octrees avec des programmes basés sur un moteur 3D commun. Le programme avec les arbres BSP basé sur le même moteur n'a pu être achevé à temps. Le programme concernant les portails a été réalisé en deux dimensions et n'utilise donc pas le moteur 3D commun. L'étude et l'implémentation de ces algorithmes a pris plus de temps que ce que nous avions prévu, et les notions de contraintes physiques abordées dans le cahier des charges comme la gravité et la gestion des collisions n'ont pu être implémentées faute de temps. La gestion des contraintes physiques nécessite en réalité une étude poussée difficilement réalisable dans le temps imparti en plus de l'étude des algorithmes de partitionnement de l'espace et pourrait à elle seule faire l'objet d'un TER.

**Bilan Personnel** : Bien que l'aspect visuel de notre travail puisse dégager un certain coté ludique, il faut garder à l'esprit le fait qu'il y a un gros travail d'étude et d'implémentation derrière tout cela. En effet notre travail a été majoritairement tourné vers des notions complexes d'algorithmique et de géométrie dans l'espace. Il nous a fallu en plus apprendre l'API Open GL pour mettre en place un moteur 3D performant et nous familiariser avec des outils de modélisation d'environnement 3D comme Blender. Tout ce travail nous a apporté des connaissances sur l'élaboration des programmes interactifs en 3D simulant un déplacement dans un environnement virtuel.

L'utilisation intensive du Wiki (voir les statistiques) a permis une bonne organisation de notre travail et nous a incité à communiquer le plus possible.

Par ailleurs, la rencontre avec Nicolas Peri (ancien étudiant en maîtrise d'informatique et développeur de jeux vidéo sur PC, X-Box et GSM), nous a appris les techniques utilisées dans les jeux actuels ainsi que les outils et les méthodes de travail dans les entreprises les développant.